A Proof Recipe for Linearizability in Relaxed Memory Separation Logic

SUNHO PARK, KAIST, South Korea
JAEWOO KIM, KAIST, South Korea
IKE MULDER, Radboud University Nijmegen, Netherlands
JAEHWANG JUNG, KAIST, South Korea
JANGGUN LEE, KAIST, South Korea
ROBBERT KREBBERS, Radboud University Nijmegen, Netherlands
JEEHOON KANG, KAIST, South Korea

Linearizability is the de facto standard for correctness of concurrent objects—it essentially says that all the object's operations behave as if they were atomic. There have been a number of recent advances in developing increasingly strong linearizability specifications for relaxed memory consistency (RMC), but scalable *proof methods* for these specifications do not exist due to the challenges arising from out-of-order executions (requiring event reordering) and selected synchronization (requiring tracking of view transfers).

We propose a proof recipe for the *linearizable history specifications* by Dang et al. in the Iris-based iRC11 concurrent separation logic in Coq. Key to our proof recipe is the notion of *object modification order (OMO)*, which generalizes the modification order of the C11 memory model to an object-local setting. Using OMO we minimize the conditions that need to be proved for event reordering. To enable proof reuse for concurrent libraries that are built on top of others, OMO provides the novel notion of a *commit-with relation* that connects the linearization points of the lower and upper libraries. Using our recipe, we verify the linearizability of the Michael–Scott queue, the elimination stack, and Folly's MPMC queue in RMC for the first time; and verify stronger specifications of a spinlock and atomic reference counting in RMC than prior work.

CCS Concepts: • **Theory of computation → Separation logic**; **Automated reasoning**; **Concurrency**.

Additional Key Words and Phrases: linearizability, relaxed memory, separation logic, automation

ACM Reference Format:
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 (June 2024), 24 pages. https://doi.org/10.1145/3656384

1 INTRODUCTION

Linearizability [Herlihy and Wing 1990] is the de facto standard specification for concurrent objects. It requires that **(1)** each operation on the object appears to take effect instantaneously at some point during its execution, called its *linearization point*; and **(2)** the linearization points of all operations form a sequential history according to their execution order, called the *linearization order*, that

Authors' addresses: Sunho Park, KAIST, Daejeon, South Korea, sunho.park@kaist.ac.kr; Jaewoo Kim, KAIST, Daejeon, South Korea, jaewoo.kim@kaist.ac.kr; Ike Mulder, Radboud University Nijmegen, Nijmegan, Netherlands, i.mulder@cs.ru.nl; Jaehwang Jung, KAIST, Daejeon, South Korea, jaehwang.jung@kaist.ac.kr; Janggun Lee, KAIST, Daejeon, South Korea, janggun.lee@kaist.ac.kr; Robbert Krebbers, Radboud University Nijmegen, Nijmegan, Netherlands, mail@robbertkrebbers.nl; Jeehoon Kang, KAIST, Daejeon, South Korea, jeehoon.kang@kaist.ac.kr.

ACM 2475-1421/2024/6-ART154
https://doi.org/10.1145/3656384

```
        d := 0; s := new_stack();
L1  d := 42;        ||  R1  try_pop(s);
L2  push(s, 1);     ||  R2  if try_pop(s) == 1:
                    ||  R3      assert(*d == 42);
```

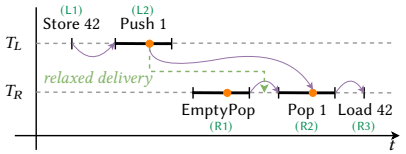


Fig. 1. An execution trace of a left (T_L) and right (T_R) thread that use a linearizable Treiber's stack in RMC. Orange dots represent linearization points and purple arrows represent the causal order.

adheres to the object's sequential specification. Linearizability of various data structures has been verified in the *sequential consistency* (SC) memory model, where all threads take turns to execute instructions on the latest state of the shared memory.

Linearizability gets complicated in *relaxed memory consistency* (RMC) models. *Relaxed behaviors*, such as instruction reordering, make the linearization order deviate from the execution order. Fig. 1 illustrates the complexities using Treiber [1986]'s stack. The global variable d is initialized with 0 and the stack s is initially empty. Assume the left thread (T_L) executes first, storing 42 to d (line L1) and pushing 1 to s (line L2). The right thread (T_R) may fail to pop a value from s (line R1) as the push event may not be visible for T_R. This greatly contrasts with the SC model where every event is immediately visible to all threads, and R1 thus cannot miss the push. Assume the event becomes visible to T_R after R1, then the pop (line R2) will succeed. This execution illustrates the following additional desired properties of the linearization order in RMC over those in the SC model:

- *Sequential specification with event reordering*: While the execution order of the stack events is L2, R1, and R2, their linearization order should be R1, L2, and R2 to observe the sequential specification of the stack: R1 (empty pop) should be linearized before L2 (push) to make sense. We call such a divergence of the linearization order from the execution order *event reordering*.
- *Causal consistency*: Despite event reordering, the linearization order should observe causality of events. For instance, L2, R2, and R1 (while satisfying the stack's sequential specification) is not a valid linearization order as it does not preserve the causally ordered events R1 and R2 that are sequentially executed in the same thread.
- *Selective synchronization*: T_R should load 42 from d (not the initial value 0) because the matching push (L2) and pop (R2) events synchronize in RMC, *i.e.*, all events observed in T_L before L2 should be visible by T_R after R2. But the synchronization is selective, *e.g.*, the non-matching push (L2) and empty pop (R1) events do not synchronize even though they are linearized.

State of the art. Various strong specifications that capture linearizability in RMC have been proposed [Batty et al. 2013; Dang et al. 2022; Dongol et al. 2018; Mével and Jourdan 2021; Mével et al. 2020; Raad et al. 2019; Singh and Lahav 2023; Smith et al. 2020]. We focus on specifications in concurrent separation logic [Brookes 2004; O'Hearn 2004] because they provide strong support for compositional verification and mechanization in a proof assistant. For the SC model, linearizability is commonly encoded through *logically atomic triples* (LATs) [da Rocha Pinto et al. 2014], which are supported by the Iris framework for higher-order concurrent separation logic in Coq [Jung et al. 2016, 2018, 2015; Krebbers et al. 2018, 2017a,b]. Linearizability of the push operation of the SC version of Treiber's stack is encoded by $\langle vs.\, \mathsf{Stack}(s, vs)\rangle\ \mathsf{push}(s, v)\ \langle \mathsf{Stack}(s, v :: vs)\rangle$. The representation predicate $\mathsf{Stack}(s, vs)$ asserts that a stack located at s has abstract state $vs \in \mathit{List}(\mathit{Val})$. The LAT expresses that $\mathsf{push}(s, v)$ *appears to transform the stack atomically* into a stack with an additional value v at some atomic instruction during push's execution, which is the linearization point. The encoding using LATs is canonical in that it coincides with the standard definition of linearizability in SC [Birkedal et al. 2021].

Mével and Jourdan [2021]; Mével et al. [2020] first use LATs to encode linearizability in RMC in the context of the Iris-based Cosmo logic for OCaml's memory model. Dang et al. [2022] show how to take this approach further, proposing *linearizable history specifications* (among others), in the context of the Iris-based iRC11 logic for ORC11 [Dang et al. 2020], a variant of the RC11 memory model [Lahav et al. 2017].[1] A key difference from LATs in the SC model is the representation predicate. The $\mathsf{Stack}(s, H)$ predicate no longer keeps track of a list of values, but it consists of the whole set H of events that have ever been performed on the stack s. Linearizability is represented as the existence of a linearization (total ordering) of H that satisfies the desired properties. When an operation is executed, the LAT specification appears to transform $\mathsf{Stack}(s, H)$ into $\mathsf{Stack}(s, H \uplus \{e\})$ atomically, where e is a new event and $A \uplus B$ is the disjoint set union of A and B.

Dang et al. [2022] propose another style of specifications, called *partial-order-based specifications*, where $\mathsf{Stack}(s, H)$ does not construct a concrete total order but only guarantees functional correctness (*e.g.*, LIFO property) among events in H. While the two styles of specifications can be equivalent in strength if partial-order-based specifications are sufficiently constrained, linearizable history specifications are more akin to the concept of linearizability and easier for clients to use thanks to the total ordering of events. As such, we focus on linearizable history specifications.

Problem. While Dang et al. [2022] show how to give strong linearizability *specifications* in relaxed memory separation logic, they did not detail how to *verify* these specifications. Moreover, for the linearizable history specifications, they verify Treiber's stack as the sole case study. This Coq proof involves a significant effort of 1,615 SLOC (significant lines, excluding comments and blank lines), which is 48 times the size of the implementation. In our experience of verifying the linearizable history specifications for other concurrent objects, we faced the following challenges:

C1 *Constructing the linearization order*: Due to event reordering, correctness of the linearization order needs to be reproved for each event. This proof should be immediate in easy cases and manageable in difficult cases. For Treiber's stack, the linearization order is straightforward as it coincides with the modification order of the head pointer, but significant bookkeeping is required. The Michael–Scott queue [Michael and Scott 1996] requires reordering, and hence Dang et al. prove only a partial-order-based specification.

C2 *Compositional verification*: Concurrent objects (*e.g.*, Folly's MPMC queue [Meta 2023]) are usually composed of smaller objects (*e.g.*, SPSC queue), and this compositional structure of implementation should ideally carry over to verification by proper library abstraction. Dang et al. conjecture that their linearizable history specifications can be compositionally verified (without case studies), but we found that non-trivial development is necessary (see §4).

C3 *Inferring the logical state*: In the SC model, the logical state of an object invariant is often straightforward to infer from the physical state. In contrast, in RMC, the connection between the logical and physical states is much less obvious due to the possibilities of event reordering. This makes interactive and semi-automated proofs in proof assistants difficult.

Contributions. We propose a **linearizability proof recipe** for concurrent objects in relaxed memory separation logic that addresses the aforementioned verification challenges.

In §3, we propose a *proof structure recipe* to address **C1** (constructing the linearization order). We observe that the linearization order usually evolves like the *modification order* (total coherence order of writes) of a shared location in the C11 memory model [Lahav et al. 2017]. We capture this observation in a novel *object modification order (OMO)* separation logic predicate. OMO enforces the linearization order to evolve only by inserting new events into the existing order. We describe

[1] Dang et al. [2022] focus on ORC11 because it is the most relaxed among practical RMC memory models. We focus on the same memory model in this paper, but we believe our recipe can be easily adapted to other memory models (see §8).

several proof rules for inserting events. These rules minimize the proof obligations needed to re-establish the desired properties of the linearization order.

In §4, we propose a *proof composition recipe* to address **C2** (compositional verification). We observe that the linearization point of an upper-level object event often coincides with that of its lower-level object event. We capture these relations using a novel *commit-with relation* inspired by the well-known simulation techniques used in proving behavior refinement [Brookes and Rounds 1983], and design proof rules that enable us to define upper-level object invariants without inspecting the lower-level object implementations. As a preliminary, we tweak Dang et al.'s specifications to expose the minimal necessary details on the events of the lower-level object, and recognize shared locations as primitive objects.

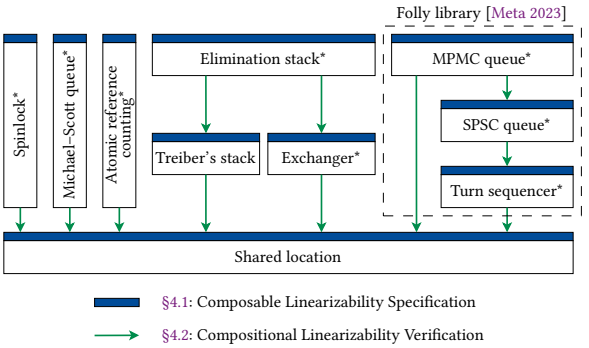


Fig. 2. Overview of our case studies (§6). *: The first verification of linearizability in RMC.

In §5, we propose a *proof automation recipe* to address **C3** (inferring the logical state). We observe that the proof rules for our structure and composition recipes minimize proof obligations so that many of them can be discharged with pattern-based automation. We adapt the Diaframe [Mulder and Krebbers 2023; Mulder et al. 2022] separation logic automation framework in Iris to RMC, and apply it to our recipes to infer the logical state in a best-effort fashion.

In §6, we demonstrate that our recipe is effective on reducing verification efforts and facilitating new proofs for RMC, see Fig. 2. We verify linearizability of Treiber's stack with 71 SLOC of Coq proof (96% reduced from Dang et al.'s proof); those of the Michael–Scott queue [Michael and Scott 1996], the elimination stack [Hendler et al. 2004], and Folly's MPMC queue [Meta 2023] for the first time; and stronger specifications of a spinlock and atomic reference counting than prior work. All our results (including the recipe itself) are mechanized using the iRC11 relaxed memory separation logic, built in the Iris framework, atop the Coq proof assistant.

In §2, we review the background. In §7 and §8, we discuss related and future work, respectively.

2 BACKGROUND

We review the necessary background about relaxed memory consistency (RMC): view-based operational models (§2.1), separation logic (§2.2), and linearizability using logically atomic triples (§2.3).

2.1 View-based Operational Models for RMC

In the sequential consistency (SC) model, each thread is guaranteed to read the latest value written to each location. This is no longer the case for RMC models. To account for the weak semantics of hardware and compiler optimizations, these models formalize out-of-order executions. RMC models constrain the values that can be read through the notions of *coherence* and *causality*. There are primarily two styles of RMC models. In axiomatic models [Batty et al. 2011; Lahav et al. 2017] coherence and causality are expressed in terms of relations between memory events in the complete execution graph. In view-based operational models, such relations are implicitly modeled by each thread's behavior depending on their thread-local view of the memory state. We review operational models, which form the semantic basis for the separation logic we use in this paper. Our explanation is simplified for brevity; details can be found in Kang et al. [2017] and Dang et al. [2020].

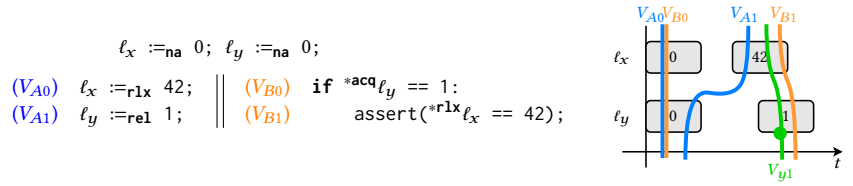


Fig. 3. Message passing with release-acquire synchronization from left thread T_A to right thread T_B.

To account for out-of-order reads, the shared memory of operational RMC models is a collection of all values written so far. Formally, a shared memory $\mathcal{M} \in \mathsf{Mem} = \mathsf{Loc} \rightharpoonup \mathsf{Time} \rightharpoonup \mathsf{Val} \times \mathsf{View}$ consists of the *location history* of messages $\mathcal{M}(\ell)$ for each location ℓ, where $A \rightharpoonup B$ is the set of partial functions from A to B. Each *message* $(v, V) = \mathcal{M}(\ell, t)$ represents that value v has been written to location ℓ at *numeric timestamp* t (see below for the *view* V).

Fig. 3 contains an example. The initial memory has messages with value 0 for locations ℓ_x and ℓ_y. If thread T_A writes 42 to ℓ_x and 1 to ℓ_y, new messages with value 42 and 1 are added to the location histories of ℓ_x and ℓ_y, respectively. Subsequently, if thread T_B is scheduled after thread T_A, it may non-deterministically read 0 (stale) or 1 (latest) from ℓ_y. However, if thread T_B reads 1 (latest) from ℓ_y, then the read from ℓ_x cannot result in a stale value, *i.e.*, it surely is 42. That is because of the *release-acquire synchronization* through the release store in T_A and acquire load in T_B.

View-based RMC models use the following notions to constrain the values that are read:

- **Coherence.** The *coherence-before* order from axiomatic models is modeled by considering the message history $\mathcal{M}(\ell)$ as totally ordered—*i.e.*, given timestamps $t_1 < t_2$, the message $\mathcal{M}(\ell, t_1)$ is *coherence-before* $\mathcal{M}(\ell, t_2)$. The X-axis in Fig. 3 represents timestamps, and initial messages have the bottom timestamp (left corner). Timestamps of different locations are not comparable.
- **Causality.** The *happens-before* order from axiomatic models is modeled through *views*. Each thread has a *current view* $V_{\mathsf{cur}} \in \mathsf{View} = \mathsf{Loc} \rightharpoonup \mathsf{Time}$, which expresses that for each location ℓ the write of $\mathcal{M}(\ell, V_{\mathsf{cur}}(\ell))$ has been observed by that thread. The current view can only increase during execution—after performing an instruction, the new current view $V_{\mathsf{cur}'}$ should extend the prior current view V_{cur}, *i.e.*, $V_{\mathsf{cur}} \sqsubseteq V_{\mathsf{cur}'}$, which is defined as $\forall \ell.\, V_{\mathsf{cur}}(\ell) \leq V_{\mathsf{cur}'}(\ell)$.

Semantics of relaxed (rlx) accesses. The semantics of a relaxed load and store operation are: **(1)** a load from ℓ reads a message at $V_{\mathsf{cur}}(\ell)$ or later in $\mathcal{M}$; **(2)** a store to ℓ writes a message at (strictly) later than $V_{\mathsf{cur}}(\ell)$ in $\mathcal{M}$; and **(3)** reading or writing a message $\mathcal{M}(\ell, t)$ updates $V_{\mathsf{cur}}(\ell)$ to t. For the example in Fig. 3, the store of 42 to ℓ_x by thread T_A updates its current view from V_{A0} to V_{A1} by incorporating the new message written to ℓ_x.

Semantics of release (rel)/acquire (acq) accesses. The semantics of a release store and an acquire load operation are similar to that of their relaxed counterparts. However, to account for release-acquire synchronization it involves view passing from the sender of the message to the receiver. This is best demonstrated by the example in Fig. 3. When thread T_A performs a release store to ℓ_y, it adds a message $(1, V_{y1})$ to the location history of ℓ_y (the written value and the updated current view). Later, when thread T_B performs an acquire load and obtains that message, its view V_{y1} is incorporated into thread T_B's current view, *i.e.*, $V_{B1} = V_{B0} \sqcup V_{y1}$, where the join $V_1 \sqcup V_2$ is defined as $\ell \mapsto \max(V_1(\ell), V_2(\ell))$. After synchronization, this ensures thread T_B reads 42 (not 0) from ℓ_x. Note that synchronization happens only when a store is release and a load is acquire (or stronger). A relaxed store operation does not put the thread's current view in the new message, and a relaxed load operation does not incorporate the message view into the thread's current view.

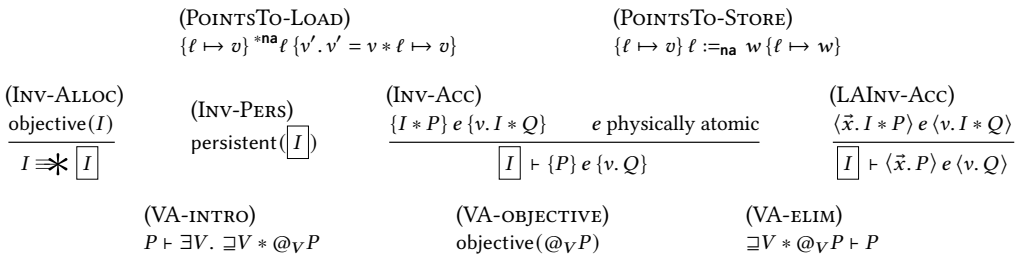


Fig. 4. Selected rules of iRC11. (The invariant rules are simplified: we omit later modalities and invariant masks, which are necessary for soundness in the general case. See Jung et al. [2018, §2.2] for details.)

Semantics of non-atomic (na) accesses. Non-atomic accesses are weaker than relaxed accesses. A data race on a non-atomic access has undefined behavior, *i.e.*, execution in the operational semantics gets stuck. Undefined behavior is modeled through a race detector [Dang et al. 2020, §5.1]. Note that the relaxed store/load on ℓ_y in Fig. 3 can be replaced by a non-atomic store/load because the release-acquire synchronization prevents data races.

2.2 Separation Logic for RMC

We present the basic principles of the iRC11 relaxed memory separation logic [Dang et al. 2020], which extends the Iris separation logic [Jung et al. 2018] with support for the ORC11 RMC model. The key difference between separation logics for SC (such as Iris) and separation logics for RMC (such as iRC11) is the necessity to account for out-of-order executions. Separation logics for SC assume that the resources of each thread and the global invariants are interpreted with respect to the latest memory state—but this principle is not sound in RMC. The resources of each thread should be interpreted with possibly stale memory states subject to the thread's current view. The iRC11 logic achieves this using *view-dependent propositions*. Fig. 4 shows selected iRC11 proof rules.

View-dependent propositions. The propositions of iRC11 are defined as view-monotone predicates $\mathit{vProp} \triangleq \mathsf{View} \xrightarrow{\mathrm{mon}} \mathit{iProp}$, where iProp is the type of Iris propositions. Monotonicity is required to ensure that a proposition remains valid when a program step increases the thread's current view. By interpreting propositions implicitly with respect to the current view, the iRC11 surface logic is similar to SC separation logic. The usual separation logic connectives (*e.g.*, quantifiers, separating conjunction) are lifted from iProp to vProp. Hoare triples $\{P\}\ e\ \{v.\ Q\}$ use view-monotone predicates for the precondition P and postcondition Q. (Like Iris, the postcondition has a binder v for the return value, which we omit if the unit value $()$ is returned.) The rules PointsTo-Load and PointsTo-Store for non-atomic accesses are exactly like their counterparts in SC separation logic. They involve the *points-to assertion* $\ell \mapsto v$, which asserts unique ownership of location ℓ with value v.

Invariants and objective resources. To share resources among threads, Iris provides the invariant connective $\boxed{I}$, which expresses that a proposition I holds at all times. Inv-Alloc says that a proposition I can be turned into an invariant $\boxed{I}$. (For simplicity, think of Iris's update $\Rrightarrow$ as an implication.) Inv-Pers says that invariants are *persistent* [Jung et al. 2018, §2.3], *i.e.*, it does not assert exclusive ownership. Persistent propositions P are duplicable ($P \dashv\vdash P * P$) and can therefore be shared among threads. Inv-Acc says a thread can get unique ownership of the resources described by I for the duration of a physically atomic step, provided that ownership is given back afterwards.

Invariants in iRC11 are more restrictive than those in Iris. The shared nature of invariants mandates their interpretation to be independent of the view, called *objective* in iRC11. This restriction is formalized by the premise objective(I) of Inv-Alloc. Crucially, the points-to assertion $\ell \mapsto v$ is

```
L1  fun push(s, v):
L2      n := new(node);
L3      n.val :=na v;
L4      loop:
L5          h := *rlx s.head;
L6          n.next :=na h;
L7          if CASrel(s.head, h, n):
L8              return ();
L9  fun try_pop(s):
L10     loop:
L11         h := *acq s.head;
L12         if h == ⊥:
L13             return ⊥;
L14         n := *na h.next;
L15         if CASacq(s.head, h, n):
L16             return *na h.val;
```

Fig. 5. An implementation of Treiber's stack in ORC11.

not objective, and thus cannot immediately be shared through invariants. To put a non-objective proposition P (such as $\ell \mapsto v$) into an invariant, VA-INTRO is used to *freeze* P with respect to the thread's current view, *i.e.*, to turn P into $\sqsupseteq V * @_V P$ for the thread's current view V. The *view-seen predicate* $\sqsupseteq V$ asserts that the thread's current view is at least V, and is formally defined as $\sqsupseteq V \triangleq \lambda V_{\mathsf{cur}}.(V_{\mathsf{cur}} \sqsupseteq V)$. The *view-at modality* $@_V P$ says that P holds at view V, and is formally defined as $@_V P \triangleq \lambda V'.\ P(V)$. Since $@_V P$ is objective by definition, it can be put in invariants and transferred to other threads. VA-ELIM allows retrieving the original resource P as soon as the thread obtains $\sqsupseteq V$, *e.g.*, by receiving it from another thread via release-acquire synchronization.

To reason about atomic operations, iRC11 provides the *atomic* points-to assertion $\ell \mapsto_{\mathrm{at}} h$, which asserts ownership of location ℓ with history h. To constrain relaxed behaviors such as reading stale values, the *history-seen observation* $\ell \sqsupseteq_{\mathrm{sn}} h'$ asserts a thread's observation of writes in the sub-history h'. For space reasons, we refer to Dang et al. [2020] for details.

2.3 Linearizability and Logically Atomic Triples in RMC

In an SC memory model, an object is *linearizable* if **(1)** each of its operations appear to take effect instantaneously at some point during the operation's execution, called the *linearization point*; and **(2)** the execution order of operations at their linearization points, called the *linearization order*, adheres to the *sequential specification* of the object. In separation logics such as Iris, linearizability is commonly formulated using *logically atomic triples* (LAT) [Birkedal et al. 2021; da Rocha Pinto et al. 2014]. These triples are of the form $\langle \vec{x}.\, P(\vec{x}) \rangle\, e\, \langle v.\, Q(\vec{x}, v) \rangle$, and express that e has an atomic instruction, called the *commit point*, which takes the precondition $P(\vec{x})$ and transforms it into the postcondition $Q(\vec{x}, v)$, where v is the return value. The LAT of the push operation of an SC stack is $\langle vs.\, \mathsf{Stack}(s, vs) \rangle\, \mathsf{push}(s, v)\, \langle \mathsf{Stack}(s, v :: vs) \rangle$, and expresses that a push of v to the stack located at s takes effect instantaneously at the linearization point, causing the abstract state to be atomically changed from vs to $v :: vs$. Since LATs allow for interference from other threads, such as concurrent calls to push and pop, the exact value of vs need not be known when the function is called. As such, the pre- and postcondition of a LAT can be bound by (a number of) quantifiers $\vec{x}$ (here, the single quantifier vs). The fact that a LAT ensures that an operation behaves as if it were atomic is witnessed by LAInv-Acc in Fig. 4, which says that invariants can be accessed around a LAT.

Out-of-order executions and event reordering. A naive adaptation of linearizability to RMC does not work due to out-of-order executions. Consider an RMC version of Treiber [1986]'s stack implemented by Dang et al. [2022] in Fig. 5.[2] The stack is represented as a linked list of nodes pointed to by a head pointer. The method `push` creates a new node, and tries to append it to the list by modifying the head pointer. The CAS (atomic compare-and-swap) loop is used to test whether a stale value for the head pointer was read, or the head pointer was changed by a concurrent push or pop, and if so tries again. The method `try_pop` tries to remove a node from the list and returns its value. If the list is empty, $\bot$ (null) is returned. The method `try_pop` also uses a CAS loop to

[2]The keen reader may notice that the CAS on L15 can be relaxed. The implementation of Dang et al. [2022] uses an acquire CAS for simplification of the proof, which we replicate here.

(Stack-Push-Spec)

$$\frac{\mathsf{SeenStack}(s, M_0) * \sqsupseteq V_0}{\langle H.\, \mathsf{Stack}(s, H)\rangle\ \texttt{push}(s, v)\ \left\langle \begin{array}{l} \exists V, M, e, E, t.\ \mathsf{Stack}(s, H \uplus \{e \mapsto E\}) * @_V\, \mathsf{SeenStack}(s, \{e\} \cup M) * \sqsupseteq V \\ \quad * e \notin \mathrm{dom}(H) * V_0 \sqsubseteq V * M_0 \subseteq M \\ \quad * E = \langle \mathsf{type} : t, \mathsf{sync} : V, \mathsf{eview} : M\rangle * \ t = \mathsf{Push}(v) \end{array} \right\rangle}$$

(Stack-Try-Pop-Spec)

$$\frac{\mathsf{SeenStack}(s, M_0) * \sqsupseteq V_0}{\langle H.\, \mathsf{Stack}(s, H)\rangle\ \texttt{try\_pop}(s)\ \left\langle v.\ \cdots * \ ((v = \bot \wedge t = \mathsf{EmptyPop}) \vee (v \neq \bot \wedge t = \mathsf{Pop}(v)))\right\rangle}$$

(Stack-Linearizable)

$$\mathsf{Stack}(s, H) \vdash \exists \pi, \sigma.\, \mathsf{interp}(H, \pi, [\,], \sigma) \wedge \mathsf{lhb}(H, \pi) \qquad \text{where } \pi : \mathbb{N}^+_{\leq |H|} \xrightarrow{1:1} \mathrm{dom}(H)$$

$$\mathsf{interp}(H, \pi, \sigma_0, \sigma) \triangleq \exists \sigma_1, \ldots, \sigma_{|H|-1}.\, \sigma_0 \xrightarrow{\pi_1, H[\pi_1]} \sigma_1 \xrightarrow{\pi_2, H[\pi_2]} \cdots \xrightarrow{\pi_{|H|}, H[\pi_{|H|}]} \sigma$$

$$\mathsf{lhb}(H, \pi) \triangleq \forall e, e'.\, e' \in H[e].\mathsf{eview} \implies \pi^{-1}(e') < \pi^{-1}(e)$$

$$\begin{array}{l} \sigma \xrightarrow{e, E} \sigma' \triangleq\ (E.\mathsf{type} = \mathsf{Init} \wedge \sigma = \sigma' = [\,]) \\ \qquad \vee\ (\exists v.\, E.\mathsf{type} = \mathsf{Push}(v) \wedge \sigma' = (e, v, E.\mathsf{sync}, E.\mathsf{eview}) :: \sigma) \\ \qquad \vee\ (\exists v, V, M, e'.\, E.\mathsf{type} = \mathsf{Pop}(v) \wedge V \sqsubseteq E.\mathsf{sync} \wedge \{e'\} \cup M \subseteq E.\mathsf{eview} \wedge \sigma = (e', v, V, M) :: \sigma') \\ \qquad \vee\ (E.\mathsf{type} = \mathsf{EmptyPop} \wedge \sigma = \sigma' = [\,]) \end{array}$$

Fig. 6. Linearizable history specification of Treiber's stack.

detect concurrent modifications. Crucially, push uses a release CAS on L7, which makes sure the thread's current view is passed to the corresponding try_pop on L11, similar to the message-passing example in §2.1. Release-acquire synchronization is also necessary to ensure that the value written at L3 is safe to read at L16 (recall, data races on non-atomic accesses have undefined behavior).

To express linearizability, we consider the events Init, Push(v), Pop(v), and EmptyPop. The commit points of Push and Pop are successful CAS operations on L7 and L15, respectively, which atomically change the linked list. The commit point of EmptyPop is on L11 in case $\bot$ is read. It is important to point out that the acquire read on L11 can read a stale value, which is the main cause of complication compared to SC. Consider the example from Fig. 1, where a thread T_L successfully performs push(s,1) on an initial stack, and another thread T_R subsequently performs try_pop(s). Now T_R may fail to read the latest value written by T_L at L11 and commit an EmptyPop event. But the execution Push(1); EmptyPop contradicts the stack's sequential specification.

Dang et al. address this problem through *linearizable history specifications*. Their key idea is to allow the linearization order to diverge from the execution order, provided causal consistency is preserved. In the example above, this means the EmptyPop event of T_R is allowed to be linearized before the Push(1) event of T_L. Fig. 6 gives the specification of Treiber's stack.[3] Compared to the SC version, the Stack(s, H) predicate no longer involves a list of values representing the abstract state. Instead, it involves a collection H of events, where each event (E) is uniquely identified by an id (e). An event consists of **(1)** type: Init, Push(v), Pop(v), EmptyPop; **(2)** sync: the *sync-view* to express synchronization; **(3)** eview: the *event-view* collection that tracks which events happened before. In the example above, the sync-view is displayed as orange dots in Fig. 1. The predicate SeenStack(s, M) tracks the collection M of event ids that have been observed by the thread.

The Stack-Push-Spec rule says that the operation adds a new event (E with fresh id e), increases the current view (from V_0 to V), and increases the thread's observation (from M_0 to M). The Stack-Try-Pop-Spec rule is similar with the exception of the highlighted area. Since the collection H is *a*

[3]We streamline Dang et al. [2022]'s version to make it amenable to generalization to other concurrent objects.

priori unordered, Stack-Linearizable says that it can be turned into a linearization order, represented as a sequence π of H's event ids, where:

(1) The interpretation predicate $\mathsf{interp}(H, \pi, [], \sigma)$ asserts that the linearization order π satisfies the sequential specification. That is, the events in H, ordered by π, transform the stack from its initial state [] to some final state σ. Here, σ is fully determined from H and π.
(2) The *local happens before* predicate $\mathsf{lhb}(H, \pi)$ asserts that the linearization order π satisfies causal consistency, *i.e.*, whenever the event e' happens before e, the event e' precedes e in π.

The interp predicate uses an abstract state transition relation $\sigma \xrightarrow{e,E} \sigma'$, which says that event E with id e transforms state σ into σ'. A state is not just the list of the values in the stack. The elements (e, v, V, M) also track the event id e of the $\mathsf{Push}(v)$ event, the sync-view V, and the observation M of the thread that performed the push. Let us explain why this bookkeeping is needed.

Selective synchronization. Linearizability in RMC is complicated further by the fact that not all events synchronize with each other. In Treiber's stack, only matching Push and Pop events are synchronized—*i.e.*, on a successful pop, only the sync-view of the thread that pushed the value is transferred. The blue parts in Fig. 6 ensure that the sync-view V and observation M of the thread that performed a push is transferred to the thread that performs the matching pop.

3 PROOF STRUCTURE RECIPE

Dang et al. [2022] show that linearizable history specifications are applicable to a variety of concurrent objects. The highlighted parts in Fig. 6 are specific to Treiber's stack, while the other parts are essentially generic. Despite the reuse of the specifications, Dang et al. did not consider reuse of proofs. For each concurrent object, they needed to build a new proof from scratch—which involves defining representation predicates such as Stack and SeenStack using ghost state, and proving that they satisfy rules such as those in Fig. 6. In this paper, we observe that a large portion of the proof follows a common structure that can be captured in a reusable library.

Our library consists of a number of separation logic predicates and proof rules to keep track of the *object modification order (OMO)*, which generalizes the *modification order* in the C11 memory model. Instead of being global to the whole memory, OMO keeps track of the coherence order of just the concurrent object in question, and therefore allows object-local reasoning. We show that to extend the OMO with a new event, it is not necessary to consider an arbitrary permutation. New events can simply be inserted into the current order, allowing us to reuse the properties of the existing OMO to prove the desired properties of the updated OMO. We present the predicates of our OMO library (§3.1), followed by the proof rules to insert events. Our proof rules are optimized to minimize proof obligations for the case of inserting a totally-ordered write (TOW) event at the end (§3.2), a read-only event in the middle (§3.3), or an arbitrary event in the middle (§3.4). Fig. 7 contains an overview of the OMO proof rules; the parts in green will be explained in §4.

The predicates of our library are defined on top of Iris's primitive constructs for ghost state. They should be considered abstract—implementation details are hidden to the user through our proof rules. Implementation details can be found in our Coq mechanization [Park et al. 2024].

3.1 Object Modification Order

Our library is parameterized over the object's *sequential specification*, which is a tuple $(T, S, \sigma_0, \xrightarrow{-,-})$ consisting of: **(1)** the *types of events* T; **(2)** the *abstract states* S; **(3)** the *uninitialized abstract state* $\sigma_0 \in S$; and **(4)** the *abstract state transition relation* $\sigma \xrightarrow{e,E} \sigma'$ between states $\sigma, \sigma' \in S$. We let $e \in \mathsf{EventId}$ and $E \in \mathsf{Event}(T) ::= \langle \mathsf{type} : t, \mathsf{sync} : V, \mathsf{eview} : M \rangle$ with $t \in T$ and $V \in \mathsf{View}$ and $M \in \wp(\mathsf{EventId})$. If it is clear from the context, we leave the parameter implicit.

Key predicates and definitions (§3.1):

(OmoAuth-Linearizable)
$$\mathsf{OmoAuth}(\gamma_o, \gamma_s, H, omo, \Sigma) \vdash \mathsf{interp}_{\mathsf{omo}}(H, omo, \sigma_0, \Sigma) \land \mathsf{lhb}_{\mathsf{omo}}(H, omo) \land \mathsf{perm}_{\mathsf{omo}}(H, omo)$$

(Omo-Compatible)
$$\mathsf{interp}_{\mathsf{omo}}(H, omo, \sigma_0, \Sigma) \land \mathsf{lhb}_{\mathsf{omo}}(H, omo) \land \mathsf{perm}_{\mathsf{omo}}(H, omo) \implies \exists(\pi : \mathbb{N}^+_{\le |H|} \xrightarrow{1:1} \mathrm{dom}(H)), \sigma.\ \mathsf{interp}(H, \pi, \sigma_0, \sigma) \land \mathsf{lhb}(H, \pi)$$

(OmoSnap-Get)
$$\frac{e \in omo[n] \qquad \sigma = \Sigma[n]}{\mathsf{OmoAuth}(\gamma_o, \gamma_s, H, omo, \Sigma) \vdash \mathsf{OmoSnap}(\gamma_o, \gamma_s, e, \sigma)}$$

(OmoAuth-OmoSnap)
$$\frac{e \in omo[n]}{\mathsf{OmoAuth}(\gamma_o, \gamma_s, H, omo, \Sigma) * \mathsf{OmoSnap}(\gamma_o, \gamma_s, e, \sigma) \vdash \Sigma[n] = \sigma}$$

(OmoAuth-Alloc)
$$\frac{E.\mathsf{eview} = \emptyset \qquad \sigma_0 \xrightarrow{e,E} \sigma}{\mathsf{Token}(\gamma_{o\ell}, e_\ell) \Rrightarrow \exists \gamma_o, \gamma_s.\ \mathsf{OmoAuth}(\gamma_o, \gamma_s, \{e \mapsto E\}, [[e]], [\sigma]) * @_{E.\mathsf{sync}}(\exists_{\gamma_o}\{e\}) * e \xrightarrow{\gamma_o, \gamma_{o\ell}} e_\ell * \mathsf{Token}(\gamma_o, e)}$$

$$\mathsf{interp}_{\mathsf{omo}}(H, omo, \sigma_{\mathsf{prev}}, \Sigma) \triangleq (omo = [\,] \land \Sigma = [\,]) \lor \left(\begin{array}{l}\exists e, \vec{e}, omo', \sigma, \Sigma'.\ omo = (e :: \vec{e}) :: omo' \land \Sigma = \sigma :: \Sigma' \\ \qquad \land\ \sigma_{\mathsf{prev}} \xrightarrow{e,E[e]} \sigma \land (\forall e' \in \vec{e}.\ \sigma \xrightarrow{e', H[e']} \sigma) \land \mathsf{interp}_{\mathsf{omo}}(H, omo', \sigma, \Sigma')\end{array}\right)$$

$$\mathsf{maxGen}(omo, M) \triangleq \max\{n \mid \exists e.\ e \in M \land e \in omo[n]\}$$

$$\mathsf{lhb}_{\mathsf{omo}}(H, omo) \triangleq \forall n, e.\ e \in omo[n] \implies \mathsf{maxGen}(omo, H[e].\mathsf{eview}) \le n$$

$$\mathsf{lin}([\vec{e}_1, \ldots, \vec{e}_n]) \triangleq \vec{e}_1 +\!\!+ \cdots +\!\!+ \vec{e}_n$$

$$\mathsf{perm}_{\mathsf{omo}}(H, omo) \triangleq \exists(\pi : \mathbb{N}^+_{\le |H|} \xrightarrow{1:1} \mathrm{dom}(H)).\ (\forall i \in \mathbb{N}^+_{\le |H|}.\ \pi(i) = \mathsf{lin}(omo)[i]) \land \mathsf{len}(\mathsf{lin}(omo)) = |H|$$

Proof rule for totally-ordered write events (§3.2):

(OmoAuth-Insert-Last)
$$\frac{e \notin \mathrm{dom}(H) \qquad \mathsf{last}(\Sigma) = \sigma \qquad E.\mathsf{eview} = M \qquad \sigma \xrightarrow{e,E} \sigma'}{\begin{array}{l}\mathsf{OmoAuth}(\gamma_o, \gamma_s, H, omo, \Sigma) * @_{E.\mathsf{sync}}(\exists_{\gamma_o} M) * \mathsf{Token}(\gamma_{o\ell}, e_\ell) \\ \Rrightarrow \mathsf{OmoAuth}(\gamma_o, \gamma_s, H \uplus \{e \mapsto E\}, omo +\!\!+ [[e]], \Sigma +\!\!+ [\sigma']) * @_{E.\mathsf{sync}}(\exists_{\gamma_o}(\{e\} \cup M)) * e \xrightarrow{\gamma_o, \gamma_{o\ell}} e_\ell * \mathsf{Token}(\gamma_o, e)\end{array}}$$

Proof rule for read-only events (§3.3):

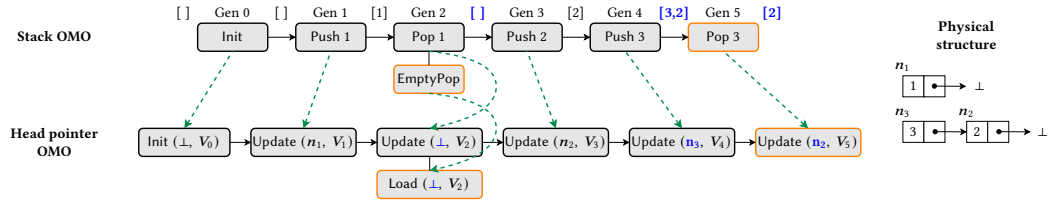
(OmoAuth-Insert-Ro)
$$\frac{e \notin \mathrm{dom}(H) \qquad E.\mathsf{eview} = M \qquad n = \mathsf{len}(omo_1) = \mathsf{len}(\Sigma_1) \qquad \sigma \xrightarrow{e,E} \sigma \qquad \mathsf{maxGen}(omo_1 +\!\!+ [\vec{e}] +\!\!+ omo_2, M) \le n}{\begin{array}{l}\mathsf{OmoAuth}(\gamma_o, \gamma_s, H, omo_1 +\!\!+ [\vec{e}] +\!\!+ omo_2, \Sigma_1 +\!\!+ [\sigma] +\!\!+ \Sigma_2) * @_{E.\mathsf{sync}}(\exists_{\gamma_o} M) * \mathsf{Token}(\gamma_{o\ell}, e_\ell) \\ \Rrightarrow \mathsf{OmoAuth}(\gamma_o, \gamma_s, H \uplus \{e \mapsto E\}, omo_1 +\!\!+ [\vec{e} +\!\!+ [e]] +\!\!+ omo_2, \Sigma_1 +\!\!+ [\sigma] +\!\!+ \Sigma_2) \\ \quad * @_{E.\mathsf{sync}}(\exists_{\gamma_o}(\{e\} \cup M)) * e \xrightarrow{\gamma_o, \gamma_{o\ell}} e_\ell * \mathsf{Token}(\gamma_o, e)\end{array}}$$

Proof rule for general write events (§3.4):

(OmoAuth-Insert)
$$\frac{e \notin \mathrm{dom}(H) \qquad E.\mathsf{eview} = M \qquad n = \mathsf{len}(omo_1) = \mathsf{len}(\Sigma_1) + 1 \qquad \mathsf{interp}_{\mathsf{omo}}(H \uplus \{e \mapsto E\}, [e] :: omo_2, \sigma, \Sigma_3) \qquad \mathsf{maxGen}(omo_1 +\!\!+ omo_2, M) < n}{\begin{array}{l}\mathsf{OmoAuth}(\gamma_o, \gamma_s, H, omo_1 +\!\!+ omo_2, \Sigma_1 +\!\!+ [\sigma] +\!\!+ \Sigma_2) * @_{E.\mathsf{sync}}(\exists_{\gamma_o} M) * \mathsf{Token}(\gamma_{o\ell}, e_\ell) \\ \Rrightarrow \exists \gamma_s'.\ \mathsf{OmoAuth}(\gamma_o, \gamma_s', H \uplus \{e \mapsto E\}, omo_1 +\!\!+ [[e]] +\!\!+ omo_2, \Sigma_1 +\!\!+ [\sigma] +\!\!+ \Sigma_3) \\ \quad * @_{E.\mathsf{sync}}(\exists_{\gamma_o}(\{e\} \cup M)) * e \xrightarrow{\gamma_o, \gamma_{o\ell}} e_\ell * \mathsf{Token}(\gamma_o, e)\end{array}}$$

Fig. 7. Selected proof rules of the OMO library.

Given a sequential specification $(T, S, \sigma_0, \xrightarrow{-,-})$, we provide a number of separation logic predicates. The key predicate is the *authoritative object modification order* $\mathsf{OmoAuth}(\gamma_o, \gamma_s, H, omo, \Sigma)$, which encodes the current linearization order. The parameter **(1)** $H \in \mathsf{EventId} \rightharpoonup \mathsf{Event}(T)$ represents the events that occurred so far; **(2)** $omo \in \mathit{List}(\mathit{List}(\mathsf{EventId}))$ describes the current order using the event ids in $\mathrm{dom}(H)$; **(3)** $\Sigma \in \mathit{List}(S)$ describes the intermediate abstract states; and **(4)** γ_o and γ_s are the *logical id* and *version id*, respectively (see §3.4 for details).



$$\mathsf{Stack}(s, H, \gamma_o, omo) \triangleq \exists \gamma_s, \gamma_{os}, \Sigma, H_s, omo_s.\ \mathsf{OmoAuth}(\gamma_o, \gamma_s, H, omo, \Sigma) * s \overset{\gamma_{os}}{\longmapsto} (H_s, omo_s) * \left(\begin{array}{l}\forall e_s \in \mathrm{dom}(H_s).\ \exists e, n, \sigma.\ e \overset{\gamma_o, \gamma_{os}}{\rightarrow} e_s * \mathsf{LocVal}(H_s[e_s].\mathrm{type}, n) \\ \qquad * \mathsf{OmoSnap}(\gamma_o, \gamma_s, e, \sigma) \\ \qquad * @_{H_s[e_s].\mathrm{sync}} (\mathsf{PhysList}(\sigma, n) * \sqsupseteq_{\gamma_o} (\{e\} \cup H[e].\mathrm{eview}))\end{array}\right) * \mathsf{CWMono}(\gamma_o, H)$$

$$\mathsf{SeenStack}(s, \gamma_o, M) \triangleq \exists \gamma_{os}, M_s.\ \sqsupseteq_{\gamma_o} M * \mathsf{SeenLoc}(s, \gamma_{os}, M_s) * \forall e \in M.\ \exists e_s \in M_s.\ e \overset{\gamma_o, \gamma_{os}}{\rightarrow} e_s$$

$$\mathsf{LocVal}(t, v) \triangleq t = \mathsf{Load}(v, \_, \_) \vee t = \mathsf{Store}(v, \_, \_) \vee t = \mathsf{Update}(\_, \_, v, \_, \_)$$

$$\mathsf{PhysList}(\sigma, n) \triangleq (\sigma = [\,] \wedge n = \bot) \vee (\exists v, n', \sigma'.\ \sigma = (\_, v, \_, \_) :: \sigma' \wedge n.\mathrm{val} \mapsto\!\square\ v \wedge n.\mathrm{next} \mapsto\!\square\ n' \wedge \mathsf{PhysList}(\sigma', n'))$$

Fig. 8. Treiber's stack representation predicates (below) with an illustration of the key invariants (above). Some Load events that are not used as commit points of Treiber's stack are omitted in head pointer's OMO structure for simplicity. The parts in orange correspond to operations discussed in §3.2 and §3.3.

The parameter *omo* is a list of lists, rather than a list, to distinguish write and read-only events. Given $omo = [\vec{e}_1, \ldots, \vec{e}_n]$ and $\Sigma = [\sigma_1, \ldots, \sigma_n]$, the head of each list of events $\vec{e}_i$ represents a write transition from σ_{i-1} to σ_i, while the elements in the tail of $\vec{e}_i$ represent a read-only transition from σ_i to itself. In §3.3 we show how distinguishing read-only events enables simpler proof rules. We call the nth element of *omo* and Σ the *nth generation*. The predicate $\mathsf{interp}_{\mathsf{omo}}(H, omo, \sigma, \Sigma)$ says that *omo* and Σ represent a valid chain of transitions starting in σ with events E, and $\mathsf{lhb}_{\mathsf{omo}}(H, omo)$ says that *omo* ensures causal consistency. The rules OmoAuth-Linearizable and Omo-Compatible collectively say that OMO correctly encodes the sequential specification and causal consistency.

The *snapshot predicate* $\mathsf{OmoSnap}(\gamma_o, \gamma_s, e, \sigma)$ is a persistent snapshot of the intermediate abstract state after interpreting the events in the linearization order up to e. OmoSnap-Get produces snapshots, while OmoAuth-OmoSnap says that e and σ match up with the Σ parameter of OmoAuth.

The rule OmoAuth-Alloc initializes the predicate OmoAuth from an initial event (e, E). The initial event is essential to support initialization parameters such as data structure size.

The *seen event predicate* $\sqsupseteq_{\gamma_o} M$ provides persistent knowledge of thread-local observation of events in the set M. When OmoAuth is initialized (OmoAuth-Alloc) or modified by inserting a new event (OmoAuth-Insert-Last, OmoAuth-Insert-Ro, OmoAuth-Insert), the new event e is observed at the sync-view of E. Observations can be merged and split, *i.e.*, $\sqsupseteq_{\gamma_o} M_1 * \sqsupseteq_{\gamma_o} M_2 \dashv\vdash \sqsupseteq_{\gamma_o} (M_1 \cup M_2)$.

Example. Fig. 8 defines the predicates $\mathsf{Stack}(s, H, \gamma_o, omo)$ and $\mathsf{SeenStack}(s, \gamma_o, M)$ for Treiber's stack. They have more parameters (γ_o and *omo*) compared to Fig. 6, which are used for library abstraction (§4.1). For now you can assume these parameters are existentially quantified.

The predicate $\mathsf{Stack}(s, H, \gamma_o, omo)$ is decomposed into **(1)** the OmoAuth predicate; **(2)** the ownership of the shared head pointer $s \overset{\gamma_{os}}{\longmapsto} (H_s, omo_s)$; and **(3)** data structure specific invariants. Instead of using iRC11's primitive atomic points-to assertion $s \mapsto_{\mathsf{at}} h$ for the head pointer, we use $s \overset{\gamma_{os}}{\longmapsto} (H_s, omo_s)$. We discuss this predicate in detail in §4.1, but for now it is sufficient to know that it equips the location s with a set H_s of event ids for Load, Store and Update (CAS) and an OMO structure omo_s. The predicate $\mathsf{SeenStack}(s, \gamma_o, M)$ is decomposed into **(1)** the seen event predicate for general OMO ($\sqsupseteq_{\gamma_o} M$) to track observations to the stack; **(2)** the seen event predicate for OMO

location ($\mathsf{SeenLoc}(s, \gamma_{os}, M_s)$) to track observations to the head pointer (§4.1); and **(3)** data structure specific invariants. The data structure specific invariants for Treiber's stack are as follows.

Stk1 Every event to the head pointer (e_s) has a corresponding event for Treiber's stack (e). This relation is established whenever e is committed at the instruction point of e_s, and is visualized by the green dotted lines in Fig. 8. This is formalized by $e \xrightarrow{\gamma_o, \gamma_{os}} e_s$, whose exact meaning is discussed in §4.2. The intermediate abstract state (σ) for the event (e) matches up with the corresponding value of the head pointer (n) and the physical linked list structure, formalized by $\mathsf{PhysList}(\sigma, n)$, The PhysList predicate is similar to the usual representation predicate for linked-lists in separation logic. The main difference is that we use the persistent points-to assertion $s \mapsto\!\!\!\square\, v$, which compared to $s \mapsto v$ is freely duplicable but forbids any operation other than load [Vindum and Birkedal 2021] (nodes of Treiber's stack are immutable).

Stk2 The green dotted lines are monotone. This is formalized by $\mathsf{CWMono}(\gamma_o, H)$, see §4.2.

Stk3 An observation of an event for Treiber's stack (e) is equivalent to the observation of the corresponding memory event (e_s). This invariant is marked in red in Fig. 8.

Representation predicates for other concurrent data structures are defined with a similar pattern: an OmoAuth predicate to govern the insertion of events, predicates to describe the state of concurrent component objects (like $s \xmapsto{\gamma_{os}} (H_s, omo_s)$), and data structure specific invariants. The key here is that OmoAuth directly captures the linearizability condition for a given data structure.

3.2 Proof Rule for Totally Ordered Write Events

The rule OmoAuth-Insert-Last allows for the insertion of a totally-ordered write (TOW) event into the OMO structure. Compared to the rule for general write events (§3.4) the premises are easy to prove. Since there is no reordering, we do not have to re-establish $\mathsf{interp}_{\mathsf{omo}}$. We only need to prove that there is a transition $\sigma \xrightarrow{e,E} \sigma'$ from the last abstract state σ with the new event E to the new state σ'. This is the minimal condition to re-establish the correctness of the linearization order. The rule transforms OmoAuth by adding $\{e \mapsto E\}$ to H, and appending $[e]$ and σ' to end of *omo* and Σ, respectively. The rule requires an observation of the event view ($\sqsupseteq_{\gamma_o} M$) with $E.\mathsf{eview} = M$, and returns an observation that includes the new event e ($\sqsupseteq_{\gamma_o} (\{e\} \cup M)$).

Example. TOW events are very common in libraries for RMC, supported by the fact that all write events in the concurrent objects we verified are totally ordered, with a few exceptions such as the Dequeue event in the Michael–Scott queue (§6.2). As an example, consider the situation in Treiber's stack in Fig. 8, where we commit a Pop(3) event (marked orange) after a successful CAS (L15 in Fig. 5). By OmoAuth-Insert-Last, we only need to prove a one-step state transition. We can deduce that the last abstract state is **[3, 2]**. The return value 3 matches with the first element of the abstract state by the invariant **Stk1** in the previous subsection. Thus, by setting the new abstract state to be **[2]**, we can prove a one-step state transition which immediately re-establishes the invariant.

3.3 Proof Rule for Read-Only Events

The rule OmoAuth-Insert-Ro allows for the insertion of a read-only event into the middle of the OMO structure. We have a special treatment for read-only events since they do not change the abstract state after taking a state transition, which allows most of the conditions on the linearization order to be reused when re-establishing it. As a result, the rule only requires one to prove **(1)** a one-step state transition that does not change the abstract state ($\sigma \xrightarrow{e,E} \sigma$), even if the new event is inserted in the middle; and **(2)** causal consistency for just the new event (marked in red). These are the minimal conditions to re-establish the correctness of the linearization order.

Example. As EmptyPop of Treiber's stack is a read-only event, OmoAuth-Insert-Ro can be used in this case. Consider the situation in Fig. 8 where we insert an EmptyPop event at the 2nd generation (marked orange) after reading $\perp$ at timestamp t_2 from the head pointer (L11 in Fig. 5). Among the two obligations, the one-step state transition can be easily proven by the invariant **Stk1** since reading $\perp$ ensures that the intermediate abstract state of the 2nd generation is empty. The second condition on causal consistency can be proven by **Stk2** and **Stk3**. If the thread has observed a later event (*e.g.*, Push(2)), it would have also observed the corresponding message (*e.g.*, at t_3), contradicting the fact that it read $\perp$ at t_2 from the head pointer. Proving only these conditions is sufficient to re-establish the invariant on the linearization order after insertion.

3.4 Proof Rule for General Write Events

The rule OmoAuth-Insert covers the general case of inserting a write event in the middle of the OMO structure. This rule is used in the verification of the Michael–Scott queue (§6.2), where a Dequeue event requires the linearization order to be reordered. Due to its generality, this rule does not preserve the correctness of the previous linearization order. A new write event can invalidate the interpretability of later events with respect to the sequential specification. Hence one needs to re-establish the chain of state transitions for all later events captured by the $\mathsf{interp}_{\mathsf{omo}}$ premise. However, causal consistency only needs to be proven for the new event (marked in red).

The invalidation of later events may invalidate snapshots taken using the predicate OmoSnap. To cope with this problem without losing persistence of the predicate, we equip OmoAuth and OmoSnap with a parameter γ_s representing the *version id*. When using OmoAuth-Insert, one gets a new version id γ_s', thereby invalidating snapshots with the old version id γ_s.

4 PROOF COMPOSITION RECIPE

Concurrent objects are often built on top of another. For example, Folly's MPMC queue [Meta 2023] is built on top of an SPSC queue, which in turn is built on top of a *turn sequencer* (see §6.3). To make verification scalable and compositional, it is crucial to observe *library abstraction*, *i.e.*, the proof of the upper-level object is done purely in terms of the specification of the lower-level object, without inspecting its implementation. To this end, we observe that the linearization point of an upper-level object event often coincides with that of its lower-level object event. We capture these relations using the *commit-with* relation of our OMO library, which is inspired by the well-known simulation techniques used in proving behavior refinement [Brookes and Rounds 1983].

We first generalize linearizable history specifications to *composable linearizability specifications*, which expose the necessary details on the events of the lower-level object to enable compositional verification of upper-level objects (§4.1). We make shared locations the lowest-level objects with corresponding composable linearizability specifications by equipping them with an OMO structure. Finally, we discuss the formal definition of the commit-with relation in our OMO library through the persistent separation logic predicate $e \xrightarrow{\gamma_o,\gamma_{o\ell}} e_\ell$ and the exclusive permission $\mathsf{Token}(\gamma_o, e)$ (§4.2). These predicates formalize the green dotted lines in the verification of Treiber's stack in Fig. 8.

4.1 Composable Linearizability Specification

The rule Stack-Try-Pop-Spec-Comp in Fig. 9 gives the composable linearizability specification for the pop operation of Treiber's stack. The differences compared to Stack-Try-Pop-Spec from §2.3 are highlighted : **(1)** the postcondition contains a permission $\mathsf{Token}(\gamma_o, e)$, which is needed to establish a commit-with relation (more details in §4.2); and **(2)** the Stack predicate exposes *omo* and includes postconditions $((omo, e) \rightsquigarrow_w omo'$ and $(omo, e) \rightsquigarrow_r omo')$ to guarantee that the OMO structure is constructed incrementally. To see how these conditions are used in compositional proofs, let us first describe the lower-level object of Treiber's stack—a shared location for the head pointer—in

Composable specification of Treiber's stack (§4.1, generalized from Fig. 6):

(Stack-Try-Pop-Spec-Comp)

$$\frac{\mathsf{SeenStack}(s, \gamma_o, M_0) * \sqsupseteq V_0}{\left\langle H,\ omo\ .\ \mathsf{Stack}(s, H,\ \gamma_o, omo\ )\right\rangle\ \texttt{try\_pop}(s)\ \left\langle \begin{array}{l} v\ .\ \cdots * \ \mathsf{Token}(\gamma_o, e)\ * \\ \quad (\ (v = \bot \land t = \mathsf{EmptyPop} \land\ (omo, e) \rightsquigarrow_r omo'\ ) \\ \quad \lor\ (v \neq \bot \land t = \mathsf{Pop}(v) \land\ (omo, e) \rightsquigarrow_w omo'\ )) \end{array} \right\rangle}$$

(Stack-Linearizable-Comp)

$$\mathsf{Stack}(s, H,\ \gamma_o, omo\ ) \vdash \exists \Sigma.\ \mathsf{interp}_{\mathsf{omo}}(H,\ omo\ ,\ [\,]\ , \Sigma) \land\ \mathsf{lhb}_{\mathsf{omo}}(H,\ omo\ ) \land\ \mathsf{perm}_{\mathsf{omo}}(H, omo)$$

$$(omo, e) \rightsquigarrow_w omo' \triangleq \exists omo_1, omo_2.\ omo = omo_1 +\!\!+ omo_2 \land omo' = omo_1 +\!\!+ [[e]] +\!\!+ omo_2$$
$$(omo, e) \rightsquigarrow_r omo' \triangleq \exists omo_1, omo_2, \vec{e}.\ omo = omo_1 +\!\!+ [\vec{e}] +\!\!+ omo_2 \land omo' = omo_1 +\!\!+ [\vec{e} +\!\!+ [e]] +\!\!+ omo_2$$

Composable specification of shared location (§4.1):

(OmoLoc-Alloc)

$$\ell \mapsto v \Rrightarrow \exists \gamma_o, V, e.\ \ell \overset{\gamma_o}{\longmapsto} (\{e\ \mapsto \langle \mathsf{type} : \mathsf{Init}(v, V), \mathsf{sync} : V, \mathsf{eview} : \emptyset\rangle\}, [[e]]) * @_V \mathsf{SeenLoc}(\ell, \gamma_o, \{e\}) * \sqsupseteq V * \mathsf{Token}(\gamma_o, e)$$

(OmoLoc-Load)

$$\frac{\mathsf{SeenLoc}(\ell, \gamma_o, M_0) * \sqsupseteq V_0 * o \neq \mathbf{na}}{\langle H, omo.\ \ell \overset{\gamma_o}{\longmapsto} (H, omo)\rangle\ {}^{*o}\ell\ \left\langle\ v\ .\ \cdots * \ t = \mathsf{Load}(v, V_r, o) \land (omo, e) \rightsquigarrow_r omo'\ \right\rangle}$$

(OmoLoc-Store)

$$\frac{\mathsf{SeenLoc}(\ell, \gamma_o, M_0) * \sqsupseteq V_0 * o \neq \mathbf{na}}{\langle H, omo.\ \ell \overset{\gamma_o}{\longmapsto} (H, omo)\rangle\ \ell :=_o w\ \left\langle \cdots * \ t = \mathsf{Store}(w, V_w, o) \land (omo, e) \rightsquigarrow_w omo'\ \right\rangle}$$

(OmoLoc-CAS)

$$\frac{\mathsf{SeenLoc}(\ell, \gamma_o, M_0) * \sqsupseteq V_0 * o \neq \mathbf{na}}{\langle H, omo.\ \ell \overset{\gamma_o}{\longmapsto} (H, omo)\rangle\ \mathsf{CAS}^o(\ell, v_r, v_w)\ \left\langle \begin{array}{l} b\ .\ \cdots * \\ \quad (\ b = \mathsf{false} \land t = \mathsf{Load}(v_{rf}, V_{rf}, \mathbf{rlx}) \land (omo, e) \rightsquigarrow_r omo' \\ \quad \lor\ b = \mathsf{true} \land t = \mathsf{Update}(e_r, v_r, v_w, V_w, o)) \land (omo, e) \rightsquigarrow_w omo' \end{array} \right\rangle}$$

(OmoLoc-Linearizable)

$$\ell \overset{\gamma_o}{\longmapsto} (H, omo) \vdash \exists \Sigma.\ \mathsf{interp}_{\mathsf{omo}}(H, omo,\ \sigma_0\ , \Sigma) \land \mathsf{lhb}_{\mathsf{omo}}(H, omo) \land \mathsf{perm}_{\mathsf{omo}}(H, omo)$$

Compositional linearizability verification (§4.2):

(OmoLe-Get)

$$\frac{e_1 \in omo[n_1] \qquad e_2 \in omo[n_2] \qquad n_1 \le n_2}{\mathsf{OmoAuth}(\gamma_o, \gamma_s, H, omo, \Sigma) \vdash \mathsf{Omo}_{\le}(\gamma_o, e_1, e_2)}$$

(OmoLe-Le)

$$\frac{e_1 \in omo[n_1] \qquad e_2 \in omo[n_2]}{\mathsf{OmoAuth}(\gamma_o, \gamma_s, H, omo, \Sigma) * \mathsf{Omo}_{\le}(\gamma_o, e_1, e_2) \vdash n_1 \le n_2}$$

$$\mathsf{CWMono}(\gamma_o, H) \triangleq \Box \forall \gamma_{o\ell}, e_\ell, e'_\ell, (e, e' \in \mathrm{dom}(H)).\ \mathsf{Omo}_{\le}(\gamma_{o\ell}, e_\ell, e'_\ell) * e \overset{\gamma_o, \gamma_{o\ell}}{\rightarrow} e_\ell * e' \overset{\gamma_o, \gamma_{o\ell}}{\rightarrow} e'_\ell * \mathrel{-\!\!*}\ \mathsf{Omo}_{\le}(\gamma_o, e, e')$$

Fig. 9. Selected proof rules of the OMO library for compositional verification.

the form of composable linearizability specification to present a unified theory for compositional proofs that applies to all lower-level objects including shared locations.

The *OMO points-to assertion* $\ell \overset{\gamma_o}{\longmapsto} (H, omo)$ and the *SeenLoc*(ℓ, γ_o, M) predicate are thin wrappers around the atomic points-to assertion $\ell \mapsto_{\mathrm{at}} h$ and the history-seen observation $\ell \sqsupseteq_{\mathrm{sn}} h'$ of iRC11. They are created from a non-atomic points-to predicate $\ell \mapsto v$ using OmoLoc-Alloc. The memory operations enjoy the composable linearizability specifications OmoLoc-Load, OmoLoc-Store, and OmoLoc-CAS. Fig. 10 shows the relation between a sample location's history and its OMO structure.

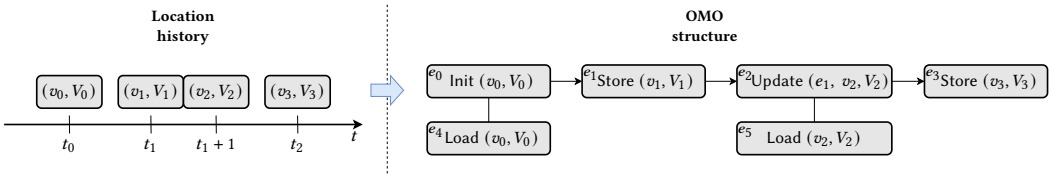


Fig. 10. Sample evolution of the history and OMO structure on a shared location. (For brevity, we omit the memory ordering o and the previous value v_r in Update events.)

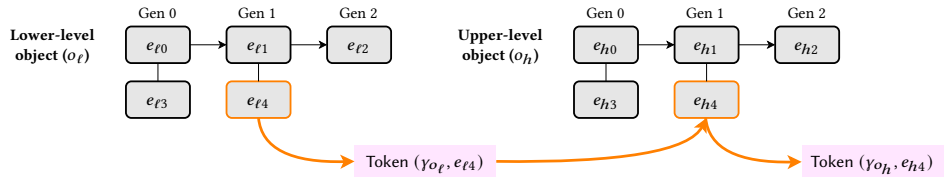


Fig. 11. Committing an upper-level object event e_{h4} at the commit point of a lower-level object event $e_{\ell 4}$.

The key to equipping shared locations with an OMO structure is their sequential specification $(T, S, \sigma_0, \xrightarrow{-,-})$. The abstract states S of shared location are of the form (e, v, V), where e is the event id of the latest message, v is its value, and V is its released view. The event types T are $\mathsf{Init}(v, V)$ for initialization, $\mathsf{Load}(v, V, o)$ for reading v, $\mathsf{Store}(v, V, o)$ for writing v, and $\mathsf{Update}(e, v_r, v_w, V, o)$ for atomically updating (*e.g.*, through a CAS) the location from value v_r to new value v_w. Event types are equipped with the acquired view V and memory ordering o. The abstract state transition relation $\sigma \xrightarrow{e,E} \sigma'$ gives the semantics of memory accesses. The most interesting case is the one for Update. A transition $(e, v, V) \xrightarrow{e',E} (e', v', V')$ requires the previous abstract state's event id e to coincide with that of the one specified in the event $E = \langle \mathsf{type} : \mathsf{Update}(e, v_r, v_w, V, o), \cdots \rangle$. This is needed to forbid future write events from being inserted between e and e'.

4.2 Compositional Linearizability Verification

To enable compositional proofs, our OMO library makes it possible to connect the linearization point of upper-level object event e to that of a lower-level object event e_ℓ. These relations are formalized using the *commit-with* predicate $e \xrightarrow{\gamma_o, \gamma_{o_\ell}\rightarrow} e_\ell$, which provides persistent knowledge of a commit-with relation between e and e_ℓ in OMOs with logical ids γ_o and γ_{o_ℓ}, respectively.

The OMO allocation rule (OmoAuth-Alloc) and insertion rules (*e.g.*, OmoAuth-Insert) illustrate that the $e \xrightarrow{\gamma_o, \gamma_{o_\ell}\rightarrow} e_\ell$ predicate is created when e is inserted into γ_o. In doing so, the rule consumes the exclusive permission $\mathsf{Token}(\gamma_{o_\ell}, e_\ell)$ for the lower-level event, and provides the exclusive permission $\mathsf{Token}(\gamma_o, e)$ for the higher-level event. The initial permission $\mathsf{Token}(\gamma_{o_\ell}, e_\ell)$ is produced by the composable linearizability specifications of the memory operations (*e.g.*, OmoLoc-Load). The rule Stack-Try-Pop-Spec-Comp exposes the token $\mathsf{Token}(\gamma_o, e)$ so that one can verify higher-level objects that use Treiber's stack as the lower-level object.

The transfer of tokens is visualized in Fig. 11. At the common linearization point for the upper-level object γ_{o_h} and lower-level object γ_{o_ℓ}, inserting the event $e_{\ell 4}$ to γ_{o_ℓ} produces $\mathsf{Token}(\gamma_{o_\ell}, e_{\ell 4})$, which is subsequently consumed to insert the event e_{h4} to γ_{o_h}, producing $\mathsf{Token}(\gamma_{o_h}, e_{h4})$ that can also be consumed by a further upper-level object atomically.

To link lower- and higher-level objects, it is often useful to ensure that their linearization orders are monotone. As demonstrated in §3.3, monotonicity of the stack invariant (**Stk2**) is used to re-establish causal consistency. The *commit-with monotonicity* predicate $\mathsf{CWMono}(\gamma_o, H)$ provides the persistent knowledge (through Iris's *persistence modality* $\square$) that the events E in the OMO with logical id γ_o are monotone w.r.t. its lower-level object. It is defined using $\mathsf{Omo}_{\leq}(\gamma_o, e_1, e_2)$, which

persistently tracks the order of the generations of events e_1 and e_2. Formally, CWMono says that any generation ordering of two upper-level events, $\mathsf{Omo}_{\leq}(\gamma_o, e, e')$, follows from that of the events projected to the same lower-level object γ_{o_ℓ}, $\mathsf{Omo}_{\leq}(\gamma_{o_\ell}, e_\ell, e'_\ell)$.

Example. In the definition of the $\mathsf{Stack}(s, H, \gamma_o, omo)$ predicate (Fig. 8), the OMO points-to assertion $s \overset{\gamma_{o_s}}{\longmapsto} (H_s, omo_s)$ is used for the head pointer, using it as a lower-level object. The green dotted lines are formalized using the commit-with predicate $e \overset{\gamma_o, \gamma_{o_s}}{\rightarrowtail} e_s$. In the proofs of the LATs for the push and pop operations, a new commit-with relation is established by consuming the head pointer's Token obtained by reading/writing to it. Monotonicity of the commit-with relation $\mathsf{CWMono}(\gamma_o, H)$ is re-established because in the case of (Push or Pop), new write events of both head pointer and Treiber's stack are appended to the end; in the case of (EmptyPop) a new EmptyPop event is inserted at the same generation with the corresponding load event on the head pointer.

5 PROOF AUTOMATION RECIPE

As the last piece of our proof recipe, we develop automation for the proof rules from §3 and §4 by building upon the Diaframe [Mulder et al. 2022] proof automation framework for Iris. We briefly review Diaframe (§5.1) and address the challenges to apply it to RMC: maintaining an up-to-date lower bound on the current thread view (§5.2), and handling invariants where the physical state of a concurrent object no longer fully determines its logical state (§5.3).

5.1 Background on Diaframe

The backbone of Diaframe is a goal-directed proof search technique—it decides the next step in the proof by looking at the current goal and the available resources in the context. For instance, when the current goal concerns a write on memory location ℓ and if a points-to assertion $\ell \mapsto v$ is in some invariant, then Diaframe opens that invariant and performs the write operation using $\ell \mapsto v$. After the operation, Diaframe tries to close the invariant to proceed onto the next step. This pattern is repeated until the proof is finished, or when the automation cannot find a way to proceed.

Closing the invariant is the main hurdle here. To see why, suppose we have an idealized invariant $\exists v, x, y.\, l \mapsto v * P(v, x) * Q(x, y)$. We can classify resources in invariants into three major categories: **(1)** *physical resources* such as $\ell \mapsto v$ that represent physical state; **(2)** *logical resources* such as ghost resources ($Q(x, y)$ in our example); **(3)** *connective resources* such as $P(v, x)$ that relate physical state (*e.g.*, the stored value v) to logical state (*e.g.*, the value x). A priori, the values x and y are not directly related to the physical value v. However, to uphold the invariant after writing a new value w to location ℓ, we must find appropriate x' and y' so that we can derive $P(w, x')$ and $Q(x', y')$ from the resources $P(v, x)$ and $Q(x, y)$. The specification of the write operation itself does not do this for us—the corresponding update rules for logical resources must be determined separately. Note that a wrong choice of existential variables v, x or y will make the goal unprovable.

Diaframe tackles this by restoring invariants *from left to right*, and having an extensible hint database of update rules. For example, if we register a hint $P(v, x) \vdash P(w, t)$ in Diaframe (where t can be determined from v, x and w) the invariant gets restored as follows in the proof:

$$\text{apply hint: } P(v,x) \vdash P(w,t) \quad \frac{Q(x,y) \vdash \exists y'.\, Q(t,y')}{P(v,x), Q(x,y) \vdash \exists x'\, y'.\, P(w,x') * Q(x',y')}$$

$$\text{choose } v' := w \quad \frac{}{\ell \mapsto w, P(v,x), Q(x,y) \vdash \exists v'\, x'\, y'.\, l \mapsto v' * P(v',x') * Q(x',y')}$$

The order of resources in the invariant is crucial for this strategy to work. It is generally advised to *place physical resources earlier than any other resources* in the invariant, so that Diaframe can infer the logical state correctly from the physical state. Additionally, Diaframe needs to be provided with hints for manipulating the connective resources P and logical resources Q.

5.2 Exposing Views in View-Dependent Predicates

Release-writes in RMC synchronize the view of the writer to subsequent acquire-readers. This is reflected in the logic as a precondition "$\exists\, V_r.\ \sqsupseteq V_r$" to release-write operations—*i.e.*, the release thread provides a lower bound for its current view, which the acquiring thread can then rely upon. It is crucial that this lower bound V_r is up-to-date enough. Proofs will otherwise get stuck, since the acquiring thread cannot provably witness required resources.

Suppose we try to prove $\exists\, V_r.\ \sqsupseteq V_r * \exists\, V_i.\ (@_{V_i} P) * (V_i \sqcup V_t \sqsubseteq V_r)$ given $P * \sqsupseteq V_t$ in our context. This goal is analogous to the proof obligation for verifying some release-writes: V_t is a lower bound on the view of our current thread, V_r is a lower bound of a view that we want to transfer to another thread by a release-write, and "$\exists V_i.\ @_{V_i} P * (V_i \sqcup V_t \sqsubseteq V_r)$" is the resource put in an invariant to transfer P, where the recipient will obtain P by using VA-ELIM after acquire-loading V_r. The hard part of this goal is that P does not necessarily hold at view V_t: the lower bound on the view of our current thread may not be up-to-date. This means we cannot just instantiate V_r with V_t: we must use VA-INTRO on P to obtain a V_i for which $@_{V_i} P * \sqsupseteq V_i$ holds, and then instantiate V_r with $V_i \sqcup V_t$.

To address this, we set up Diaframe to *always* use VA-INTRO to all view-dependent resources before each access to shared memory. This discipline makes sure we always have an up-to-date lower bound on the view of our thread around every memory access, which in turn ensures that we have sufficient information in the logic to verify release-writes. We will thus be able to prove the goal discussed above: before performing release-write, we directly apply VA-INTRO to P to obtain a V_i for which $@_{V_i} P * \sqsupseteq V_i$ holds, then merge $\sqsupseteq V_i$ with $\sqsupseteq V_t$ to $\sqsupseteq (V_i \sqcup V_t)$.

Example. The proof of push of Treiber's stack crucially relies on up-to-date lower bounds for views. The linearization point of push is a successful release-CAS (L7 in Fig. 5), which synchronizes the writes to the (non-atomic) value and next fields, to the acquire-read in L11 in the pop function. We ensure that this part of the proof goes through automatically: the non-atomic points-to assertions of the value and next fields go through VA-INTRO, and the joined view is sent to the acquire-reader.

5.3 Inferring Logical State

Recall from §5.1 that Diaframe usually infers the logical state from the physical state after symbolically executing instructions. However, the connection between physical and logical states is much less obvious in RMC. Consider the existentially quantified variables in Stack (Fig. 8). Unlike SC, where we only keep the most recent state, in RMC we are required to keep all past states (*i.e.*, inside H and Σ) and maintain invariants for all of them. This makes inference of the logical state more difficult, especially in situations where a new event must be inserted before a past state. Even worse, incorrect choices for such variables can cause the proof to reach a stuck state, requiring the proof engineer to manually intervene and backtrack to the point of the incorrect choice.

We designed the proof rules in §3 and §4 carefully to reduce the search space of existential variables for Diaframe to explore as follows. First, we constrain the evolution of the linearization order by requiring new events to be inserted only into the current OMO (§3.1). This constrains the possibilities for choosing a new logical state—it is often enough to choose the correct insertion rule to correctly infer the logical state. In addition, our proof rules require only minimal proof obligations to re-establish the correctness of OMO, decreasing the overhead of closing the invariant. Notably, one does not need to reprove linearizability after every change to the logical state: linearizability always follows from OMOAUTH-LINEARIZABLE. This is in contrast with partial-order-based specifications [Batty et al. 2013; Raad et al. 2019] that allow more possibilities on the logical state, and have a larger number of proof obligations.

Second, we use the OMO-based specifications of operations on lower-level objects (§4.1) as a guide for updating the upper-level object. After a write or read-only operation on a lower-level

object, a corresponding $\mathsf{Token}(\gamma_o, e)$ is added to the proof state. Our hints detect such Tokens, and update the logical state accordingly by choosing the corresponding insertion rule for the composite structure. For example, following the observation that the linearization point of a lower-level object usually corresponds to an event of the same type (*i.e.*, write or read-only) of the upper-level object, the presence of $\mathsf{Token}^W(\gamma_o, e)$ (indicating that (e) is a write event), triggers the hint for insertion of a write event (OmoAuth-Insert-Last).[4] Similarly, $\mathsf{Token}^R(\gamma_o, e)$ indicates a read-only event (e), and triggers the hint for OmoAuth-Insert-Ro. Moreover, we ensure our hints extend the proof state appropriately, *e.g.*, on insertion of a new event (e) into the OMO, a commit-with relation is added. These can be matched with later proof obligations as demonstrated in the example below.

Third, we ensure that Diaframe stops at the key moments in the proofs where manual reasoning is necessary. For instance, inferring the correct sync-view for Folly's MPMC queue is extremely non-trivial since it is dependent on synchronization done *after* the linearization point. To these existentially quantified variables, we add a marker—either in the invariant or in the postcondition of the specification. This marker tells Diaframe to stop the proof automation, and explicitly *ask for* a witness, whenever it would otherwise try to infer a value. Another point of interest is possible linearization points. Due to the complex nature of the OMO predicates, it is difficult for Diaframe to automatically recognize the linearization point. Thus, we make Diaframe stop before attempting to close the invariant, and ask the user whether the linearization point happens now. This approach enables a fruitful collaboration between interactive proofs and proof automation.

Example. We illustrate how these ideas work together during the verification of the Treiber stack's pop. Specifically, we consider the insertion of an EmptyPop event, which happens directly after reading $\bot$ in L11 (Fig. 5), producing a location event e_s. Treiber's stack invariant is in Fig. 8.

(1) Before attempting to restore the stack invariant (*i.e.*, $\mathsf{Stack}(s, H, \ldots)$), Diaframe stops and asks whether the linearization point happens now or later. In the case of EmptyPop, linearization happens *now*, and so the prover tells Diaframe to commit to the linearization point.

(2) The specification Stack-Try-Pop-Spec-Comp dictates the insertion of the EmptyPop event. Specifically, it prescribes a transition from $\mathsf{Stack}(s, H, \ldots)$ to $\mathsf{Stack}(s, H \uplus \{e \mapsto \ldots\}, \ldots)$ for a fresh event id $e \notin \mathrm{dom}(H)$. In the proof, this means we must make OmoAuth do a corresponding transition, and show the Stack predicate holds for these arguments.

(3) Diaframe performs this transition automatically: it finds a hint that applies OmoAuth-Insert-Ro. This hint is triggered by the Token^R resource, obtained after symbolically executing the load instruction with OmoLoc-Load. The hint additionally creates related resources, including the event mapping $e \xrightarrow{\gamma_o, \gamma_{o_s}} e_s$, OmoSnap for the new event e, and monotonicity of the event mapping with the new event included ($\mathsf{CWMono}(\gamma_o, H \uplus \{e \mapsto \ldots\})$).

(4) To create the new Stack predicate, from left to right, Diaframe instantiates the OmoAuth obtained from (**3**), and the physical obligation $s \xmapsto{\gamma_{o_s}} (H_s \uplus \{e_s \mapsto \ldots\}, \ldots)$ in the proof context.

(5) The third line of the stack predicate ($\forall e_s' \in \mathrm{dom}(H_s \uplus \{e_s \mapsto \ldots\}).\ \ldots$) is partially instantiated for old events in H_s by re-using the corresponding part from the old Stack predicate. Proof obligations remain for the new e_s, which has three existentially quantified variables e, n, and σ. e is automatically instantiated by the resource $e \xrightarrow{\gamma_o, \gamma_{o_s}} e_s$ obtained in step (**3**). n can be inferred by the location event e_s's type, and σ can be inferred by the OmoSnap also obtained in step (**3**). PhysList is trivial since we have $n = \bot \land \sigma = []$.

(6) Finally, Diaframe uses the CWMono predicate that we just obtained from (**3**).

[4]There is no hint for the general insertion rule OmoAuth-Insert since it is too complicated for automation. Instead, the user can pause automation, manually apply the rule, and resume automation.

Table 1. Quantitative analysis in SLOC of the Coq proofs (excluding empty lines and comments). "N/A" in Comparison column: previous work does not exist. "N/A" in OMO + Diaframe column: we leave it as future work due to performance problems in Coq.

Object	OMO	OMO+Diaframe	Comparison	
Treiber's stack	386 (- 76%)	71 (- 96%)	1,615	(Linearizable history specification)
Spinlock	322 (+374%)	83 (+ 22%)	68	(Weaker specification)
Michael–Scott queue	1,319 (- 41%)	613 (- 73%)	2,246	(Partial-order-based specification)
Turn sequencer (Folly)	259	136	N/A	
SPSC queue (Folly)	403	154	N/A	
MPMC queue (Folly)	690	282	N/A	
Exchanger	399 (- 67%)	147 (- 88%)	1,219	(Partial-order-based specification)
Elimination stack	841 (- 51%)	436 (- 75%)	1,721	(Partial-order-based specification)
Atomic reference counting	4,786 (+117%)	N/A	2,202	(Weaker specification)

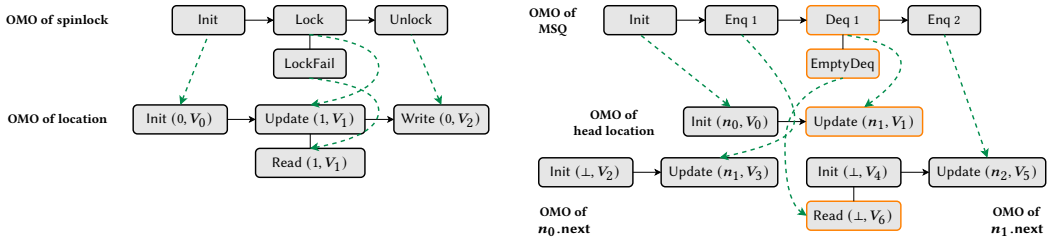


Fig. 12. OMO structures and commit-with relations of a spinlock (left) and the Michael–Scott queue (right).

6 CASE STUDIES

Table 1 compares, for several RMC objects, the line counts of (**OMO**) our manual proof with OMO; (**OMO+Diaframe**) our automated proof with OMO and Diaframe; and (**Comparison**) the existing proof in separation logic for RMC. Using the OMO structure reduces the proof effort by 57%, while the use of Diaframe additionally reduces the proof effort by 58% (overall 81% reduction) in comparison to prior work.[5] All our predicates and rules for OMO are formalized in Coq with 10,243 SLOC, while Diaframe's hint databases are also formalized in Coq with 4,404 SLOC.

6.1 Spinlock

We verify the linearizability of a spinlock with the following operations: **(1)** `new_lock()` allocates a spinlock, producing an Init event; **(2)** `try_lock`(ℓ) tries to acquire the lock at ℓ, producing either a Lock or a LockFail event; and **(3)** `unlock`(ℓ) releases the lock, producing an Unlock event. Fig. 12 illustrates that the commit-with relation between a spinlock and its location's events is bijective (like for Treiber's stack in Fig. 8). The linearizability specification we verify is stronger than the version of Dang et al. [2020]: their specification of `try_lock` does not rule out spurious failures in a single thread. Safety of the following can only be proved with our specification:

$$\ell \leftarrow \texttt{new\_lock}(); \ \texttt{assert}(\texttt{try\_lock}(\ell)) \ \texttt{/* must succeed */}; \ \texttt{unlock}(\ell).$$

[5]Verifications of spinlock and atomic reference counting are not considered when comparing proof efforts with ours since they are verified for weaker specifications by Dang et al. [2020].

6.2 Michael–Scott Queue

We verify the linearizability of the Michael–Scott queue (MSQ) [Michael and Scott 1996] with the following operations: **(1)** `new_queue()` allocates a queue, producing an Init event; **(2)** `enq`(q, v) enqueues a value v to the queue at q, producing an Enq v event; and **(3)** `try_deq`(q) tries to dequeue from the queue at q, producing a Deq v or an EmptyDeq event. The only existing verification of MSQ in RMC separation logic is done for the partial-order-based specification [Dang et al. 2022].

Fig. 12 illustrates the commit-with relation from MSQ. Unlike Treiber's stack, multiple lower-level objects (including the head and next field of each node) participate in the commit-with relation. Remember that write events on different locations may be reordered, preventing these events from having a total order. For example, consider a situation where a thread sequentially performs two `try_deq`(q) operations without observing an Enq 2 event, producing Deq 1 and EmptyDeq as presented in Fig. 12. In this case, we should not add Deq 1 to the end of OMO structure, otherwise we cannot incrementally construct OMO structure when inserting an EmptyDeq event. In general, to account for possible EmptyDeq events, Deq events are inserted in the middle by OmoAuth-Insert.

6.3 Folly's MPMC Queue

We verify the linearizability of the MPMC queue and its components from the Folly library [Meta 2023]. This case study demonstrates that **(1)** our proof composition recipe (§4) works even for a nested composition from location to "turn sequencer", to SPSC queue, and finally to MPMC queue; and **(2)** our proof structure recipe (§3) supports *external linearization points* that are determined by external factors like other threads. The Deq operation has an external linearization point, which is delayed until the matching Enq event is committed. Our recipe supports helping by sending the component Token and an *atomic update* (representing the obligation to find a linearization point) to another thread, which then consumes it and establishes the commit-with relation.

6.4 Elimination Stack

We verify the linearizability of the elimination stack [Hendler et al. 2004] as a composition of the Treiber's stack and exchanger. This case study also demonstrates the effectiveness of our proof composition recipe (§4) and support of external linearization points. The elimination stack has the same specification as Treiber's stack, but its events can be actually reordered unlike Treiber's stack.

6.5 Atomic Reference Counting

We verify the linearizability of a simplified version of Rust's atomic reference counting (Arc) library verified by Dang et al. [2020]. This case study showcases the scalability of our proof recipe to complex libraries with **(1)** a variety of functionalities, totaling 14 specifications for 11 functions; **(2)** complex synchronization patterns, *e.g.*, collecting distributed ownerships along with thread views to destroy the object in the end; and **(3)** release and acquire fences. As in the case of the spinlock, our specification is stronger than that of Dang et al. in that ours can rule out spurious failures in most realistic cases while the latter cannot. For example, safety of the following example (simplified for presentation purpose; see [Park et al. 2024]) can only be proved with our specification:

$\ell \leftarrow$ `new_arc()`; `assert(drop_arc(`ℓ`))`; `/* must succeed in deallocation */`.

6.6 Clients

To ensure that our specification is strong enough, we verify several programs such as the Treiber's stack client shown in Fig. 1, the spinlock client presented in §6.1, and the Arc client presented in §6.5. Recall from §1, §6.1 and §6.5 that these programs are not provable with weaker (but simpler) specifications that exhibit spurious behaviors.

7 RELATED WORK

Verification in relaxed memory separation logic. Dang et al. [2020] develop the iRC11 relaxed memory separation logic to verify the Rust type system and its standard libraries. They consider several concurrent objects, including a spinlock and an Arc. Being concerned with the type safety instead of functional correctness, they use weaker specifications than ours (§6.1, §6.5).

Building on top of iRC11, Dang et al. [2022] verify Treiber's stack, the Michael–Scott queue, and the elimination stack w.r.t. several specifications encoded through LATs. Our compositional linearizability specifications (Fig. 9) further develop their linearizable history specifications. We improve their verifications of the linearizable history specification of Treiber's stack, and the partial-order-based specifications of the Michael–Scott queue and the elimination stack (§1).

Mével et al. [2020] develop the Cosmo separation logic for the multicore OCaml memory model, in which Mével and Jourdan [2021] verify a concurrent bounded queue specified using an LAT. Their work influenced the aforementioned work by Dang et al. [2022] and thus ours. However, their work is based on OCaml memory model, which is less relaxed than C11 (and thus ORC11).

Verification of concurrent objects in RMC. Several strong correctness conditions for concurrent objects have been verified directly on low-level RMC semantics [Batty et al. 2013; Dongol et al. 2018; Raad et al. 2019; Singh and Lahav 2023; Smith et al. 2020]. Compared to this work, most of these works do not consider modular client reasoning or mechanization in a proof assistant.

Raad et al. [2019] propose partial-order-based specifications that support more relaxed, non-linearizable objects such as a weak version of the Herlihy–Wing queue [Herlihy and Wing 1990; Raad et al. 2019]. In contrast, our total-order-based specifications are tailored for more tight objects.

Batty et al. [2013]; Dongol et al. [2018]; Singh and Lahav [2023] advocate for the use of contextual refinement as the correctness condition for RMC libraries to provide a complete library abstraction for clients. Contextual refinement coincides with linearizability in the SC model [Filipović et al. 2010], but it remains future work to investigate if this is the case in RMC too.

Automated verification. Existing work on automated verifications of RMC libraries does not target strong specifications such as linearizability, while automated verifications of linearizability mostly focus on the SC memory model.

GenMC [Kokologiannakis et al. 2019] is a model checker parameterized over various memory models including RC11. Given a complete program, it enumerates all possible executions and checks such properties as code assertions, data race freedom, and liveness of spin loops. GenMC does not support compositional specifications of libraries as it targets closed programs only.

Summers and Müller [2018] encode the RSL [Vafeiadis and Narayan 2013], FSL [Doko and Vafeiadis 2016], and FSL++ [Doko and Vafeiadis 2017] relaxed memory separation logics into the Viper tool for automated deductive verification [Müller et al. 2017]. They verify Hoare-style specifications of several libraries including Arc and RWSpinlock, but do not consider stronger specifications such as linearizability. They exploit single-location invariants to automate invariant access when the corresponding location is accessed. However, to verify sophisticated concurrent objects (such as the Michael–Scott queue and Folly's MPMC queue) we rely on Iris/iRC11 invariants that involve multiple locations, which are not supported by these encodings in Viper.

In the context of the SC model, Voila [Wolf et al. 2021] is a proof outline checker for LATs in the TaDA logic [da Rocha Pinto et al. 2014], built on top of the Viper framework. Being a proof outline checker, Voila is not fully automated—annotations need to be added for every key step. This is in contrast with fully-automated tools such as Cave [Vafeiadis 2010] and Poling [Zhu et al. 2015] based on shape analysis. Full automation comes at the cost of compositional verification and limits the range of concurrent objects that can be verified. Plankton [Meyer et al. 2022, 2023] is a recently

developed linearizability checker based on the Flow framework [Krishna et al. 2018] that extends the range of supported objects while retaining automation. Plankton performs temporal interpolation to prove the linearization of concurrent search structures with minimal user annotations. None of the aforementioned tools for the SC model are foundational—neither their soundness (meta theory) nor their implementation has been verified in a proof assistant. Diaframe [Mulder and Krebbers 2023; Mulder et al. 2022] addresses this problem by building a tool with strong automation on top of Iris in Coq. Diaframe is extensible with user-defined hints, and has been used to automatically verify various properties in the SC model, including LATs. Its extensibility has been key for us to extend it with support for the iRC11 logic and linearizability in RMC. To the best of our knowledge, we are the first to design a framework for linearizability in RMC that is amenable to automation.

8 CONCLUSION AND FUTURE WORK

Our proof recipe addresses a problem of scalability that arises in verifying concurrent objects in RMC by encapsulating complexities of relaxed memory in simple abstractions. Our key observation is threefold: the linearization order of a concurrent object usually evolves like the modification order of a shared location (§3); for library abstraction, the linearization points of an upper-level object event often coincides with that of its lower-level object event (§4); and even for RMC, many proof obligations can be discharged with pattern-based automation (§5). As future work, we will apply our recipe to more diverse and complex objects in more realistic settings as follows.

Hindsight reasoning. We expect our recipe to support *hindsight reasoning*, even without additional reasoning principles such as prophecy variables [Jung et al. 2020]. Hindsight reasoning has the common proof pattern of generating an event early in the time and checking whether the event is valid later in retrospect. This is non-trivial in the SC setting, because only the latest state is maintained in the invariant. We expect this to be easier in our recipe for relaxed memory, as it maintains the entire history of events in the OMO structure of the invariant.

Other RMC memory models. We expect that our recipe can be adapted to other RMC memory models for the following reasons. **(1)** The core ideas of the OMO structure and the commit-with relation are not specific to iRC11 but generally applicable to linearizability in RMC. Essentially, the OMO structure streamlines the reasoning of event reordering (§3) and the commit-with relation exposes the minimal information to observe library abstraction (§4). **(2)** The C11 memory model (on which ORC11 and iRC11 are based) has arguably the most relaxed semantics among practical RMC memory models. The synchronization mechanisms in the OCaml memory model [Dolan et al. 2018] (using atomic locations) and Java memory model [Bender and Palsberg 2019] (using volatile and VarHandle) can be encoded with various C11 memory access modes.

Other styles of specifications. We expect that a large portion of our development can be readily reused in proving other styles of specifications (*e.g.*, partial-order-based specifications). One can use the OMO structure with a trivial state transition system (*i.e.*, $\forall \sigma, \sigma', e, E.\ \sigma \xrightarrow{e,E} \sigma'$ holds). By doing so, one can enjoy all the proof rules (§3, §4) and existing automation supports (§5). However, one might need to design a sophisticated invariant with less help from the OMO library, and additional hints for Diaframe that facilitate automatic re-establishment of the invariant.

ACKNOWLEDGMENTS

We thank the PLDI 2024 reviewers for their valuable feedback. Sunho Park, Jaewoo Kim, Jaehwang Jung, Janggun Lee, and Jeehoon Kang are supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT2201-06.

DATA AVAILABILITY STATEMENT

The Coq development for this paper can be found in [Park et al. 2024].

REFERENCES

Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library Abstraction for C/C++ Concurrency. In *POPL*. 235–248. https://doi.org/10.1145/2480359.2429099

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *POPL*. 55–66. https://doi.org/10.1145/1926385.1926394

John Bender and Jens Palsberg. 2019. A formalization of Java's concurrent access modes. *PACMPL* 3, OOPSLA, Article 142 (2019). https://doi.org/10.1145/3360568

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. https://doi.org/10.1145/3473586

Stephen D. Brookes. 2004. A Semantics for Concurrent Separation Logic. In *CONCUR (LNCS, Vol. 3170)*. 16–34. https://doi.org/10.1007/978-3-540-28644-8_2

Stephen D. Brookes and William C. Rounds. 1983. Behavioural equivalence relations induced by programming logics. In *ICALP (LNCS, Vol. 154)*. 97–108. https://doi.org/10.1007/BFB0036900

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. https://doi.org/10.1007/978-3-662-44202-9_9

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

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. https://doi.org/10.1145/3519939.3523451

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

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

Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding data races in space and time. In *PLDI*. 242–255. https://doi.org/10.1145/3192366.3192421

Brijesh Dongol, Radha Jagadeesan, James Riely, and Alasdair Armstrong. 2018. On abstraction and compositionality for weak-memory linearisability. In *VMCAI (LNCS, Vol. 10747)*. 183–204. https://doi.org/10.1007/978-3-319-73721-8_9

Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for concurrent objects. *TCS* 411, 51 (2010), 4379–4398. https://doi.org/10.1016/j.tcs.2010.09.021

Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A scalable lock-free stack algorithm. In *SPAA*. 206–215. https://doi.org/10.1145/1007912.1007944

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

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269. https://doi.org/10.1145/2951913.2951943

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. https://doi.org/10.1017/S0956796818000151

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). https://doi.org/10.1145/3371113

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. https://doi.org/10.1145/2775051.2676980

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

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model Checking for Weakly Consistent Libraries. In *PLDI*. 96–110. https://doi.org/10.1145/3314221.3314609

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *PACMPL* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (LNCS, Vol. 10201)*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

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

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

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

Meta. 2023. Folly: Facebook Open-source Library. https://github.com/facebook/folly

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). https://doi.org/10.1145/3473571

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

Roland Meyer, Thomas Wies, and Sebastian Wolff. 2022. A Concurrent Program Logic with a Future and History. *PACMPL* 6, OOPSLA2, Article 174 (2022). https://doi.org/10.1145/3563337

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

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

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

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

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2017. Viper: A verification infrastructure for permission-based reasoning. 104–125. https://doi.org/10.3233/978-1-61499-810-5-104

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR (LNCS, Vol. 3170)*. 49–67. https://doi.org/10.1007/978-3-540-28644-8_4

Sunho Park, Jaewoo Kim, Ike Mulder, Jaehwang Jung, Janggun Lee, Robbert Krebbers, and Jeehoon Kang. 2024. Artifact for "A Proof Recipe for Linearizability in Relaxed Memory Separation Logic", PLDI 2024. https://doi.org/10.5281/zenodo.10933398

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). https://doi.org/10.1145/3290381

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

Graeme Smith, Kirsten Winter, and Robert J. Colvin. 2020. Linearizability on Hardware Weak Memory Models. *Form. Asp. Comput.* 32, 1 (2020), 1–32. https://doi.org/10.1007/s00165-019-00499-8

Alexander J. Summers and Peter Müller. 2018. Automating Deductive Verification for Weak-Memory Programs. In *TACAS (LNCS, Vol. 10805)*. 190–209. https://doi.org/10.1007/978-3-319-89960-2_11

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

Viktor Vafeiadis. 2010. Automatically Proving Linearizability. In *CAV (LNCS, Vol. 6174)*. 450–464. https://doi.org/10.1007/978-3-642-14295-6_40

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

Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael–Scott queue (proof pearl). In *CPP*. 76–90. https://doi.org/10.1145/3437992.3439930

Felix A. Wolf, Malte Schwerhoff, and Peter Müller. 2021. Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA. In *FM (LNCS, Vol. 13047)*. 407–426. https://doi.org/10.1007/978-3-030-90870-6_22

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2015. Poling: SMT Aided Linearizability Proofs. In *CAV (LNCS, Vol. 9207)*. 3–19. https://doi.org/10.1007/978-3-319-21668-3_1

Received 2023-11-16; accepted 2024-03-31