Compass: Strong and Compositional Library Specifications in Relaxed Memory Separation Logic

Hoang-Hai Dang
MPI-SWS
Germany
haidang@mpi-sws.org

Jaehwang Jung
KAIST
South Korea
jaehwang.jung@kaist.ac.kr

Jaemin Choi
KAIST
South Korea
jaemin.choi98@kaist.ac.kr

Duc-Than Nguyen
University of Illinois at Chicago
USA
dnguye96@uic.edu

William Mansky
University of Illinois at Chicago
USA
mansky1@uic.edu

Jeehoon Kang
KAIST
South Korea
jeehoon.kang@kaist.ac.kr

Derek Dreyer
MPI-SWS
Germany
dreyer@mpi-sws.org

Abstract

Several functional correctness criteria have been proposed for *relaxed-memory* consistency libraries, but most lack support for *modular client reasoning*. Mével and Jourdan recently showed that *logical atomicity* can be used to give strong modular Hoare-style specifications for relaxed libraries, but only for a limited instance in the Multicore OCaml memory model. It has remained unclear if their approach scales to weaker implementations in weaker memory models.

In this work, we combine logical atomicity together with *richer partial orders* (inspired by prior relaxed-memory correctness criteria) to develop stronger specifications in the weaker memory model of Repaired C11 (RC11). We show their applicability by proving them for multiple implementations of stacks, queues, and exchangers, and we demonstrate their strength by performing multiple client verifications on top of them. Our proofs are mechanized in Compass, a new framework extending the iRC11 separation logic, built atop Iris, in Coq. We report the first mechanized verifications of relaxed-memory implementations for the exchanger, the elimination stack, and the Herlihy-Wing queue.

CCS Concepts: • **Theory of computation → Separation logic**.

Keywords: C11, relaxed memory models, separation logics, linearizability, logical atomicity, Iris

ACM Reference Format:
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 *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3519939.3523451

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523451

1 Introduction

Reasoning about concurrent programs is notoriously hard, and relaxed-memory consistency (RMC) makes that hardness all the more notorious. In contrast to the traditional sequential consistency (SC) model [49] where threads take turns accessing shared memory, RMC models have to include various tricky semantic features to account for both multicore hardware *and* compiler optimizations—typically executing instructions out-of-order. As such, the formal semantics of RMC models themselves require extensive ongoing research [5, 6, 11, 24, 42, 48, 51, 60, 61, 74], and program verification against the low-level model semantics has been mostly performed on small programs, such as litmus tests.

To support higher-level and compositional verification of complex RMC programs as well as *libraries*, several concurrent separation logics (CSLs) have been proposed for RMC [17, 22, 23, 30, 41, 54, 68, 72, 73]. These logics have been applied to verify tricky RMC algorithms such as locks, stacks, queues, read-copy-update [69], and reference counting [23], and even to verify soundness of the Rust type system under RMC [17]. However, these works (except Cosmo [54]—see more below) verify implementations only against some "reasonable" specifications that are sufficient for their respective purposes, but do not necessarily capture the libraries' full functional correctness. For example, the queue specification proven in iGPS [41, 72] captures only the fact that a dequeue is synchronized with the enqueue that it is matched with—it does *not* ensure the standard FIFO property of queues.

Thus, stronger functional correctness CSL specifications (hereafter, *specs*) for RMC libraries are needed, especially for clients that build new libraries out of existing ones, relying on certain strong properties of the existing libraries to verify the new library's implementation. In this work, we explore several strong CSL specs for various RMC data structure

types and demonstrate how these specs support *modular client reasoning* in a new framework we call Compass.

1.1 Strong and Compositional Functional Correctness for RMC Libraries

An omnipresent challenge in RMC verification is the fact that, unlike in the SC setting, there is no canonical way to specify full functional correctness of a library that may expose relaxed behaviors. While *linearizability* [34] is the de facto standard correctness condition for concurrent libraries, it does not extend to many highly concurrent libraries, including those in RMC: these libraries tend to have less synchronization or control, and it may be that a *linearization* is extremely difficult to construct (*e.g.*, Herlihy-Wing queue [34]) or that the library has no useful sequential behaviors (*e.g.*, exchangers [31, 63]). Therefore, various linearizability-like criteria have been proposed as alternatives [3, 9, 10, 20, 29, 33, 36, 57], especially for relaxed memory [4, 8, 21, 25, 26, 35, 45, 62]. These works essentially share one basic idea in relaxing linearizability: instead of requiring a *total order* on a library's operations, one requires only that operations respect some *partial orders*. These works, however, have little support for modular client reasoning. In this paper, we aim to improve the proposed relaxations of linearizability with *Hoare-style* specs to support better modular reasoning about clients who rely on strong correctness guarantees of RMC libraries.

Accordingly, we take as our starting point one of the key proof techniques for achieving strong specs *and* modular client reasoning in (SC) CSLs: *logical atomicity* [13, 39, 40, 67]. Logically atomic specs are similar to standard Hoare-style specs, except that they additionally provide the abstraction that the specified operation takes effect *atomically*. In particular, they give the client *atomic access* to the exact, up-to-date *abstract state* of the data structure at the moment in time when the operation occurs, thus enabling the client to build a concurrent protocol governing how the data structure is used (how the abstract state may evolve). If the client wants to compose multiple data structures, they can build a more complex protocol governing multiple abstract states, all the while enjoying the benefits of separation logics.

Logical atomicity has been applied mostly in the SC setting, and only recently did Mével and Jourdan [53] demonstrate its use to give stronger CSL specs for RMC libraries. Unsurprisingly, the application of the technique needs to account for relaxed behaviors: Mével and Jourdan needed to combine logical atomicity with the tracking of some *synchronization* information among library operations, reminiscent of the partial orders from the relaxations of linearizability. But they only needed limited synchronization tracking, because their logic, Cosmo [54], is sound only for the Multicore OCaml memory model [24], and they only gave one spec for a concurrent queue and verified one client.

To see concretely the limitations of Cosmo, consider the example in Figure 1, which shows a Message-Passing (MP)

```
enq(q, 41);  ||  deq(q)  ||  while (*acq flag == 0){};
enq(q, 42);  ||           ||  deq(q)
flag :=rel 1 ||           ||  // return 41 or 42, not empty
```

Figure 1. A Message-Passing (MP) client with Queues

client of queues in a weaker memory model. Here, the queue is accessed concurrently by 3 threads: the left-most thread performs 2 enqueues (**enq**), the middle one performs a dequeue (**deq**), and the right-most thread waits for the signal by the left-most thread through flag and then performs a dequeue. A weak implementation of dequeue can return *empty* even though the queue is not empty, due to contention. However, in this example, the right-most thread *cannot* get an empty dequeue result, because (1) at most one enqueue could have been consumed concurrently by the middle thread, and (2) due to the release-acquire synchronization through flag, the thread has synchronized with the two enqueues.

Unfortunately, the Cosmo spec only exposes *internal* (to the implementation) synchronizations among operations, without taking into account how additional *external* synchronizations created by the client (such as the synchronization through flag) can affect the behaviors of dequeues. It therefore *cannot* exclude the possibility that the right-most thread's dequeue returns empty.

1.2 Contributions

In this paper, we generalize Mével and Jourdan's approach by combining **logical atomicity** with **richer partial orders** inspired by the relaxations of linearizability, so that we can give stronger specs for more weakly consistent libraries, in a more relaxed memory model. But, given the plethora of partial orders from those relaxations of linearizability, which one should we use? We believe the *event-graph* based criteria proposed by Raad et al. [62] ("Yacovet") are the most general, because in that framework a verifier can give a library stronger or weaker specs by choosing the partial orders they prefer and by stating suitable library-specific *consistency conditions* on the partial orders. Therefore, in this work, we decided to encode Yacovet criteria in our separation logic and enhance them further with logical atomicity.

We evaluate the flexibility of this approach with several styles of specs. First, we combine the Cosmo-style specs—which we call the $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{so}}$ style (§2.3) because it tracks abstract states and the *synchronized-with* (so) relation between operations—with Yacovet-inspired event-graphs to track the larger *happens-before* (hb) relation. We call this the $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{hb}}$ spec style (§3.1), which suffices to verify the MP example in Figure 1. We then consider the $\mathsf{LAT}_{\mathsf{hb}}$ (§3.2) and $\mathsf{LAT}^{\mathsf{hist}}_{\mathsf{hb}}$ (§3.3) styles, a weakening and a strengthening of $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{hb}}$, respectively. $\mathsf{LAT}^{\mathsf{hist}}_{\mathsf{hb}}$ strengthens $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{hb}}$ with a *linearizable history*

to give tighter specs for stronger implementations. Meanwhile $\mathsf{LAT}_{\mathsf{hb}}$ abandons abstract states so as to be satisfiable by weaker implementations, and is the most faithful encoding of Yacovet criteria. We demonstrate the strength, satisfiability, and support for client reasoning of our specs with multiple mechanized libraries and client verifications.

Our technical contributions are as follows.

- We develop Compass, a new specification framework built atop the iRC11 separation logic [17], which is sound for the ORC11 [17] memory model—a variant of RC11 [48] that has non-atomic, release-acquire, and relaxed accesses, and fences, and forbids load-buffering behaviors, *i.e.*, $\mathsf{po} \cup \mathsf{rf}$ is acyclic.
- As in Cosmo, specs in Compass reuse the general definition of logically atomic triples from Iris. However, to state useful specs and verify implementations against them, we need several other extensions to iRC11 (§5): *objective invariants*, *view-explicit modalities*, and *atomic points-to assertions*. These constructs exist in simple forms in Cosmo, but for the weaker memory model of ORC11 we need a more extensive interface for them, and correspondingly, a more intricate model to establish their soundness.
- With Compass, we give strong functional specs, in the styles mentioned above, for a variety of library types, including queues (§3.1, §3.2), stacks (§3.3), and exchangers (§4.2). In the context of RMC separation logics, our exchanger specs are the first ever proposed, while our other specs are stronger than existing ones.
- We verify several implementations of stacks, queues, and exchangers against their corresponding specs. We demonstrate the usefulness and compositionality of our specs through several client verifications, including a verification of an RMC elimination stack (§4.1) that composes a stack and an exchanger modularly, relying solely on their Compass specs.
- All of our specs, library verifications, and client verifications, as well as the Compass framework and the iRC11 extensions, are mechanized in Iris [37, 38, 40, 44], in Coq.[1] We report the first mechanized RMC verifications of exchanger [63], elimination stack [32], and the Herlihy-Wing queue [34]. While these verifications required significant manual effort, their sizes suggest that they are still manageable: our library verifications are between 1.5KLOC and 3.0KLOC long, with a median of 2.1KLOC, while our client verifications are between 0.1KLOC and 0.5 KLOC long, with a median of 0.2KLOC.

In the interest of space and comprehensibility, we do not present in detail all of our contributions. Instead, after reviewing some background in §2, we present instances of our specs for queues in §3 and the compositional verification of the elimination stack in §4. In §5, we briefly discuss the extensions to iRC11 needed by Compass. We conclude with related and future work in §6.

[1] Available as supplementary materials accompanying this paper [18].

2 Background: Separation Logic Specs for Strong Memory Models

Strong memory models provide strong guarantees about the ordering of memory operations, making it easier to write clearly correct library implementations. Weaker (more relaxed) memory consistency models offer more opportunities for more efficient implementations, which, on the other hand, may provide weaker guarantees to clients. In this section we review existing specs in stronger memory models, and in §3 we will present several of our specs in the weaker ORC11 model, with the Queue data structure as an example (Figure 2). We review, in §2.1, the traditional Hoare-triple-based specs for sequential queues; in §2.2, *logical atomicity* [13, 39, 67] and its uses to give strong specs for concurrent SC queues; and in §2.3, how Cosmo [53] extends those specs for RMC with *thread views*.

2.1 Sequential Specifications for Queues

The separation logic sequential specs for queues are given as Seq-Enq and Seq-Deq (Figure 2). Program logics typically give specs for a program e as *Hoare triples* of the form $\{P\}\ e\ \{Q\}$, where P is called the *precondition* and Q the *postcondition*. The intuitive interpretation of a triple is that if the program state satisfies P before the execution of e, then after e finishes executing, the state satisfies Q.[2] Assertions like P and Q specify either properties of the current *global* program state (in traditional Hoare logic), or ownership of *parts* of the state required for the code to run (in separation logic).

For example, Seq-Enq specifies that an enqueue function call $\mathbf{enq}(q, v)$ can run safely as long as it has $\mathsf{Queue}(q, vs)$, an abstract separation logic assertion that represents full ownership of the queue object q (an instance of the data structure). An implementation can define $\mathsf{Queue}(q, vs)$ as arbitrary resources that it specifically needs. But from the perspective of clients, $\mathsf{Queue}(q, vs)$ is abstract because it asserts that q's current state can be seen *abstractly* as a list of values vs—that is, the queue's elements are currently vs, ordered by the list order. Seq-Enq then says that $\mathbf{enq}(q, v)$ requires and consumes q's ownership at the beginning of the call, and at the end of the call it returns the ownership with the updated abstract state $vs +\!\!+ [v]$, reflecting the operation's effects: v has been enqueued to the end of q. Conversely, by Seq-Deq, a dequeue $\mathbf{deq}(q)$ also consumes q's ownership and, if the queue is not empty, returns the head value v of vs and gives back the ownership with only its tail vs'. (The notation

[2] In this paper, we focus on *partial correctness*, where the triple interpretation only requires that Q holds afterwards *if* e terminates. We do not yet consider *total correctness*, where e is also required to terminate. Our partial correctness does, however, ensure that e is *safe* to execute.

$\{v.\, Q\}$ denotes the postcondition as a predicate over the returned value v.) Otherwise, if q is empty, **deq**(q) returns empty (ϵ) and the fact that the abstract state—both before and after the operation—is empty ($vs = [\,]$).

That an operation is allowed to consume the queue ownership for the *whole* duration of its execution is what makes the specs *sequential*: a group of threads cannot access the ownership Queue(q, vs) concurrently in order to perform concurrent enqueues and/or dequeues. To have strong specs for such *fine-grained concurrency*, we need logical atomicity.

2.2 SC Specifications with Logical Atomicity

In fine-grained concurrency, a concurrent object's ownership is shared for concurrent accesses, and contention is most commonly resolved by *atomic* read-modify-write (RMW) instructions, such as compare-and-swap (CAS). In this case, even if a concurrent object's operation involves multiple steps of computation, it "takes effect" atomically during a single one of those steps. This is the intuition of *logical atomicity*: from the perspective of clients, the operation *appears* to be atomically updating the object exactly around a single atomic instruction—often called the *commit* or *linearization* point of the operation.

As such, a client should need to provide ownership of the concurrent object only at the operation's commit point, and can expect the update to happen right after that point. This idea is encoded in *logically atomic triples* (LATs) [13, 39, 40, 67], of the form $\langle P\rangle\ e\ \langle Q\rangle$, with angle brackets $\langle\ \rangle$ instead of curly braces. The intuitive interpretation is also a bit more subtle than normal Hoare triples: $\langle P\rangle\ e\ \langle Q\rangle$ means that there exists a commit point (instruction) c by which e *atomically* consumes P, transforms it, and returns Q.

Using LATs, we can give strong specs like SC-Enq and SC-Deq (Figure 2) to fine-grained concurrent SC queues. Here we use red font-face to denote the gradual changes in the specs. One obvious change is the aforementioned angle brackets $\langle\ \rangle$. Less obvious is the quantification of vs in the precondition $\langle vs.\ \text{Queue}(q, vs)\rangle$: this is a special form of universal quantification that signifies the possibility that the queue may be modified concurrently. Specifically, it signifies that during the specified enqueue/dequeue operation, other threads may be changing the state vs of the queue *arbitrarily*, up until the commit point of the operation, when it atomically updates the state to what is described in the postcondition. For example, SC-Enq says that **enq**(q, v) can withstand arbitrary concurrent updates to the state vs of q, up until the commit point when it atomically transforms Queue(q, vs) (where vs is the state at that instant) to the new state Queue$(q, vs +\!\!+ [v])$. In contrast, the sequential spec Seq-Enq implicitly quantifies over vs with a normal universal quantifier ($\forall vs$) at the outside: this allows the implementation to assume exclusive ownership of Queue(q, vs) for an arbitrary but *unchanging* vs, thereby prohibiting concurrent interference.

Last but not least, we add a *local* precondition isQueue(q), another abstract assertion that encodes *persistent* separation logic facts about the queue, *e.g.*, facts about its head and tail pointers. These facts are persistent in the sense that they are freely duplicable, and they are local in the sense that they are to be provided at the beginning of a call, so that operations can use them for the whole execution, more conveniently than Queue(q, vs) which is neither duplicable nor local.

Intuitively, it should be clear that $\langle P\rangle\ e\ \langle Q\rangle$ is a stronger spec than $\{P\}\ e\ \{Q\}$, seeing as the former permits concurrent interference whereas the latter does not. But how does a client actually make use of these LATs to arbitrate concurrent accesses to a shared resource like Queue(q, vs)? To that end, we need one more ingredient from CSLs: *invariants*.

Logical atomicity and invariants. Invariants can be seen as logical, global spaces where resources can be stored for concurrent accesses. The catch is that accesses must be *(physically) atomic—i.e.*, take place during a single step of computation—and invariants must be re-established after each access, so that they indeed hold *invariantly* (*i.e.*, after each step). The standard access rule for invariants is given in Inv-Acc: a physically atomic instruction c can access and rely on I, in addition to P, for its execution, as long as it restores I afterwards. The assertion $\boxed{I}$ asserts the existence of I in the global invariant space. (The "later" modality $\triangleright$ is an artifact of the step-indexed model of Iris, which we will gloss over.)

$$\textsc{Inv-Acc}\ \frac{\{\triangleright I * P\}\ c\ \{\triangleright I * Q\} \qquad c\ \texttt{atomic}}{\boxed{I} \vdash \{P\}\ c\ \{Q\}} \qquad \textsc{LAInv-Acc}\ \frac{\langle\triangleright I * P\rangle\ e\ \langle\triangleright I * Q\rangle}{\boxed{I} \vdash \langle P\rangle\ e\ \langle Q\rangle}$$

The LAT invariant access rule LAInv-Acc *strengthens* Inv-Acc, as it relaxes the restriction of "accessing around atomic instructions" to "accessing around *logically* atomic expressions". With this rule, clients can build protocols to use and combine libraries with LAT specs. For example, with an invariant that ties together two queues by a relation R, *i.e.*, $\boxed{\exists vs_1, vs_2.\ \text{Queue}(q_1, vs_1) * \text{Queue}(q_2, vs_2) * R(vs_1, vs_2)}$, we can use LAInv-Acc with SC-Enq and SC-Deq to verify clients that use the two queues and adhere to the "protocol" R. For example, R may require that vs_1 and vs_2 are disjoint, or even more specifically, that one queue contains only odd numbers and the other contains only even numbers.

In summary, with logical atomicity and invariants, one can give stronger modular specs for fine-grained concurrent libraries. Furthermore, LAT specs can be seen as giving *abstract operational semantics* to a library's operations. As such, the library should be linearizable, *i.e.*, there is a total order of its operations according to which the concurrent object appears to behave sequentially. In fact, Birkedal et al. [7] recently showed formally that, in SC, logical atomicity implies linearizability. It is therefore an important tool to achieve full functional correctness *and* modular client reasoning.

2.3 RMC Specifications with Views

However, linearizability and logical atomicity do not directly extend to relaxed memory. In RMC, a total order of operations (the linearization) might not exist, or if it does exist, it may not be very useful. In contrast to the SC model where every atomic instruction is *synchronized with* every other atomic instruction, in RMC an atomic instruction may only be synchronized with *some* other instructions. It is the partially-ordered synchronizations—formally defined as the *happens-before* (hb) relation—between operations that really matter for their correctness, not the total order. In the terms of logical atomicity, this means that an update to the state by the commit of an operation o may only be meaningful to operations that are synchronized with o. Consequently, LAT specs for RMC libraries have to additionally account for hb. To see how to write these specs, we need to introduce per-thread *views*, an approximation of hb that is typically found in operational semantics and program logics for RMC memory models [17, 41, 42, 47, 51, 54, 59, 61, 66, 68].

Views: an approximation of happens-before. The idea of views comes from the fact that in RMC, threads may observe the effects of writes to physical memory locations differently, depending on what kind of memory instructions they have performed. To model such differences, each thread is equipped with a *local* view, often formally defined as a map from memory locations to timestamps: $\mathit{View} ::= \mathit{Loc} \to \mathit{Time}$. The timestamps are indices into an ordering of the writes to a location.[3] A thread's local view records its *observations*—the writes to memory that the thread has observed, *e.g.*, synchronized with. By performing memory instructions, a thread updates its local view (its observations), and it performs synchronizations by *sending its view to other threads*. This can be seen more concretely in the following Compass rules (simplified) for release writes and acquire reads.[4]

Rel-Write

$$\{\sqsupseteq V * \ell \mapsto h\}\ \ell :=_{\mathtt{rel}} v \left\{ \begin{array}{l} (). \exists t \notin h, V'.\, V(\ell) < t * \\ \quad V \sqcup \{\ell \mapsto t\} \sqsubseteq V' * \\ \quad \sqsupseteq V' * \ell \mapsto h[t \mapsto (v, V')] \end{array} \right\}$$

Acq-Read

$$\{\sqsupseteq V * \ell \mapsto h\}\ {}^{*\mathtt{acq}}\ell \left\{ \begin{array}{l} v. \exists t.\, V(\ell) \le t * h(t) = (v, V') * \\ \quad \sqsupseteq (V \sqcup V') * \ell \mapsto h \end{array} \right\}$$

Both rules concern (1) a persistent fact $\sqsupseteq V$ (read "seen V") that the executing thread π's current local view is *at least* V, and (2) the *atomic points-to* ownership $\ell \mapsto h$ of the location ℓ that the thread is writing to/reading from. The atomic points-to includes a *history* h ($\in \mathit{Time} \xrightarrow{\text{fin}} \mathit{Val} \times \mathit{View}$) of ℓ that, unlike the traditional separation logic *points-to* ($\ell \mapsto v$), is a set of write events that may still be visible to some threads, and that are ordered by the timestamp order. Rel-Write says that a release write extends the history h with a new element (v, V') at a fresh timestamp t. The view V' is the thread π's view after the instruction, as encoded in $\sqsupseteq V'$, and V' *includes* the view V before the instruction and the timestamp t of the write itself. The *view inclusion* relation is a partial order on views that is derived from the timestamp order, formally $V_1 \sqsubseteq V_2 ::= \forall \ell.\, V_1(\ell) \le V_2(\ell)$. Furthermore, V' is also the view of the write event inserted into the history (as in $h[t \mapsto (v, V')]$), reflecting the semantic behavior that π *releases* its observations (its local view V) through the write. Another thread, say ρ, can perform an acquire read from that write event, and by Acq-Read, *acquires* the write event view V' into its local view, as in $\sqsupseteq (V \sqcup V')$. As such, the release-acquire synchronization between π's release write and ρ's acquire read is reflected in the logic by π's sending its view V to ρ. Intuitively, any operation that happens-before π's release write is observed in V', and therefore also observed by ρ's acquire read.

The release-acquire rules demonstrate how view transfers approximate the *synchronized-with* (so) relation, the part of hb that records *inter*-thread synchronizations. The other part of hb is the *program order* (po) relation that records the *intra*-thread order, and is approximated in view inclusion by the fact that a thread's view only grows as it runs.

Views and view inclusion are a useful abstraction of hb and have formed the backbone of several CSLs for RMC [17, 41, 54, 68]. These logics use views mainly to prove the soundness of their rules, and try to *hide* views at the user level as much as possible to regain the simplicity of traditional SC logics. However, hiding views weakens the logics, and views appear to be inevitable in order to achieve strong LAT specs, as Mével and Jourdan demonstrate with their Cosmo specs.

Cosmo specs for queues. Abs-So-Enq and Abs-So-Deq (in Figure 2) are a simplified version of Cosmo specs for multi-producer multi-consumer queues. They differ from the SC specs in the extra tracking of views (in red in Figure 2): (1) the specs take the "seen view" assertion $\sqsupseteq V$ as a *local precondition* (that is, outside of the LAT precondition and needed at the beginning of the call); and (2) the abstract state is no longer just a list of values, but a list of value-view pairs, where the view component of a pair is the view of the enqueue operation (after its commit point). Similar to the release-acquire rules, the views in the abstract state support view transfers between matching enqueue-dequeue pairs: by Abs-So-Enq, an enqueue releases its local view V at its commit point, and by Abs-So-Deq, the matching dequeue acquires V into its local view, also at its respective commit point. Effectively, they expose the so relation between matching enqueue-dequeue pairs via views in the abstract state. This

[3] Timestamps are typically just natural numbers, but can be more complex depending on the memory model. And depending on the complexity of the model or the operations, a thread may also need *several* local views.

[4] Both rules are given in normal Hoare triples, but they can also be given in LATs, because the instructions are physically atomic.

SEQ-ENQ
$\{\mathsf{Queue}(q, vs)\}\ \mathbf{enq}(q, v)\ \{().\ \mathsf{Queue}(q, vs +\!\!+ [v])\}$

SEQ-DEQ
$\{\mathsf{Queue}(q, vs)\}\ \mathbf{deq}(q)\ \{v.\ (vs = [\,] * v = \epsilon * \mathsf{Queue}(q, [\,])) \lor (\exists vs'.\ vs = v :: vs' * \mathsf{Queue}(q, vs'))\}$

SC-ENQ
$\mathsf{isQueue}(q) \vdash \langle vs.\ \mathsf{Queue}(q, vs)\rangle\ \mathbf{enq}(q, v)\ \langle().\ \mathsf{Queue}(q, vs +\!\!+ [v])\rangle$

SC-DEQ
$\mathsf{isQueue}(q) \vdash \langle vs.\ \mathsf{Queue}(q, vs)\rangle\ \mathbf{deq}(q)\ \langle v.\ (vs = [\,] * v = \epsilon * \mathsf{Queue}(q, [\,])) \lor (\exists vs'.\ vs = v :: vs' * \mathsf{Queue}(q, vs'))\rangle$

ABS-SO-ENQ
$\mathsf{isQueue}(q) * \sqsupseteq V \vdash \langle vs.\ \mathsf{Queue}(q, vs)\rangle\ \mathbf{enq}(q, v)\ \langle().\ \mathsf{Queue}(q, vs +\!\!+ [(v, V)])\rangle$

ABS-SO-DEQ
$\mathsf{isQueue}(q) * \sqsupseteq V \vdash \langle vs.\ \mathsf{Queue}(q, vs)\rangle\ \mathbf{deq}(q)\ \langle v.\ (v = \epsilon * \mathsf{Queue}(q, vs)) \lor (\exists vs', V'.\ vs = (v, V') :: vs' * \mathsf{Queue}(q, vs') * \sqsupseteq V')\rangle$

ABS-HB-ENQ
$\mathsf{SeenQueue}(q, G_0, M_0) * \sqsupseteq V \vdash$
$\langle G, vs.\ \mathsf{Queue}(q, vs, G)\rangle\ \mathbf{enq}(q, v)\ \left\langle \begin{array}{l}().\ \exists G' \sqsupseteq G, M' \supseteq M_0, V' \sqsupseteq V.\ \mathsf{Queue}(q, vs +\!\!+ [(v, V')], G') \\ \quad * \mathsf{SeenQueue}(q, G', M') * \sqsupseteq V' * \exists e \notin G.\ e \in M' \land G' = G\big[e \mapsto (\mathsf{Enq}(v), V', M')\big]\end{array}\right\rangle$

ABS-HB-DEQ
$\mathsf{SeenQueue}(q, G_0, M_0) * \sqsupseteq V \vdash$
$\langle G, vs.\ \mathsf{Queue}(q, vs, G)\rangle\ \mathbf{deq}(q)\ \left\langle \begin{array}{l} v.\ \exists vs', G' \sqsupseteq G, M' \supseteq M_0, V' \sqsupseteq V.\ \mathsf{Queue}(q, vs', G') * \mathsf{SeenQueue}(q, G', M') * \sqsupseteq V' \\ \quad * \lor \left\{ \begin{array}{l} (v = \epsilon \land vs' = vs \land \exists d \notin G.\ d \in M' \land G' = G\big[d \mapsto (\mathsf{Deq}(\epsilon), V', M')\big]) \\ (\exists V_e.\ vs = (v, V_e) :: vs' \land \exists e, M_e, d \notin G.\ G(e) = (\mathsf{Enq}(v), V_e, M_e) \land (e, \_) \notin G.\mathsf{so} \land V_e \sqsubseteq V' \\ \quad \land M_e \cup \{e, d\} \subseteq M' \land G' = G\big[d \mapsto (\mathsf{Deq}(v), V', M')\big] \land G'.\mathsf{so} = \{(e, d)\} \cup G.\mathsf{so}) \end{array}\right. \end{array}\right\rangle$

ABS-HB-QUEUE-CONSISTENCY
$\mathsf{Queue}(q, vs, G) \vdash \mathsf{QueueConsistent}(vs, G)$

$V \in \mathit{View} ::= \mathit{Loc} \to \mathit{Time}$
$e \in \mathit{EventId} ::= \mathbb{N}$
$\mathit{QueueEvent} ::= \mathsf{Enq}(v) \mid \mathsf{Deq}(v) \mid \mathsf{Deq}(\epsilon)$
$M \in \mathit{LogView} ::= \wp(\mathit{EventId})$
$\mathit{Event} ::= \mathit{QueueEvent} \times \mathit{View} \times \mathit{LogView}$
$G \in \mathit{Graph} ::= (\mathit{EventId} \to \mathit{Event}, \wp(\mathit{EventId} \times \mathit{EventId}))$

$\mathsf{QueueConsistent}(vs, G) :=$
$\land \left\{ \begin{array}{ll} \forall (e, d) \in G.\mathsf{so}.\ \exists v.\ G(e).\mathsf{type} = \mathsf{Enq}(v) \land G(d).\mathsf{type} = \mathsf{Deq}(v) \land \ldots & (\textsc{Queue-Matches}) \\ \forall (e, d) \in G.\mathsf{so}, e'.\ G(e').\mathsf{type} = \mathsf{Enq}(\_) \to (e', e) \in G.\mathsf{lhb} \to & \\ \quad \exists d'.\ (e', d') \in G.\mathsf{so} \land (d, d') \notin G.\mathsf{lhb} & (\textsc{Queue-FIFO}) \\ \forall d, e.\ G(d).\mathsf{type} = \mathsf{Deq}(\epsilon) \to G(e).\mathsf{type} = \mathsf{Enq}(\_) \to & \\ \quad (e, \_) \notin G.\mathsf{so} \to (e, d) \notin G.\mathsf{lhb} & (\textsc{Queue-EmpDeq}) \\ \ldots & \end{array}\right.$

Figure 2. Specifications of Queue operations, from stronger to weaker memory consistency models.

is why we call them $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{so}}$ style. (The complete Cosmo specs also track so among enqueues and among dequeues.)

Abstract state and read-only operations. However, by using just the abstract state, the specs do not specify behaviors of *read-only* operations that do not modify the abstract state. For example, in ABS-SO-DEQ, a failing empty dequeue is a read-only operation, and the $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{so}}$ specs do not give us any new facts about *vs*. This is weaker than in the SC model, where SC-DEQ says that dequeues fail with ϵ only if the state *vs* is truly empty (at the commit point).

Realistically, an RMC spec cannot be quite as strong as the SC spec: recall that in RMC effects can appear to threads differently, so it may be that the thread π sees the queue as empty and returns ϵ, but the queue is in fact not empty, because a fresh enqueue by another thread ρ has not become visible to π yet. But we can do better than the empty case of ABS-SO-DEQ, which gives the client no useful information. In the next section (§3), we present specs that expose more of the hb relation, enough to cover read-only operations such as failing dequeues. Using those specs, we can verify the MP client in Figure 1: by combining the queue's richer hb relation with the client's *external* hb relation, we prove that the right-most thread's dequeue cannot return empty.

3 Richer Partial Orders for Stronger Specs in a Weaker Memory Model

We now present several of our logically atomic specs that, by exposing richer partial orders that can be combined with *external synchronizations*, can stay reasonably strong and yet still satisfiable by more relaxed implementations in the weaker ORC11 memory model. In §3.1 we present the $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{hb}}$ style which generalizes the $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{so}}$ style, and its instance for

queues, which suffices to verify the MP client in Figure 1. In §3.2 and §3.3, we present the $\mathsf{LAT}_{\mathsf{hb}}$ and $\mathsf{LAT}^{\mathsf{hist}}_{\mathsf{hb}}$ spec styles, a weakening and a strengthening of $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{hb}}$, respectively.

3.1 Graph-Based Specs to Encode Partial Orders

The $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{hb}}$ style extends the $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{so}}$ style by exposing a greater part of hb. An instance for queues is given in Abs-Hb-Enq, Abs-Hb-Deq, and Abs-Hb-Queue-Consistency (Figure 2). That these specs are stronger than those of Cosmo can be seen easily by ignoring the added red parts. The main improvement of this instance is in Abs-Hb-Deq's failure case, where the caller sees the queue as empty. Here, the spec provides more information about how the resulting read-only *empty dequeue* operation is ordered with other operations in hb.

As read-only operations have no effects on the abstract state, we need a new component G to identify and relate them to other operations. The component $G \in \mathit{Graph}$ is a general construction inspired by the declarative specs of Yacovet [62]. Yacovet works on *whole-program execution graphs*, and abstracts them into per-library *event graphs* of operations, where every operation is uniquely identified by an event. A Yacovet spec for a library encodes the ordering between events in a graph as partial orders that must satisfy some *library-specific consistency conditions*. Here, we encode Yacovet specs with the event graph component G. The main differences with Yacovet are that (1) G records only the library events that have happened so far, not complete executions; and (2) our specs are stated as separation logic LATs, so each operation can access the current, up-to-date event graph G and only needs to extend G with the operation's event and to maintain the graph's consistency.

The (simplified) types of event graphs are given in the bottom left of Figure 2. A graph G is a pair of (1) a function that maps each event id $e \in \mathit{EventId}$ to event data of type Event, and (2) a set of event id pairs that encodes the so relation. We use $G(e)$ to denote the event data for e in G, and $G.\mathsf{so}$ to denote the so relation of G.

The type Event is a tuple of (1) an event type (`type`), (2) a *physical* view (`view`), and (3) a *logical* view (`logview`). In Figure 2 we give an instance of the event type for queues: the events can be an *enqueue* event of v ($\mathsf{Enq}(v)$), a *successful dequeue* event of v ($\mathsf{Deq}(v)$), or a *failing (empty) dequeue* event ($\mathsf{Deq}(\epsilon)$). An event's physical view is the view at the commit point of the operation that the event represents, and is needed in the logic to interact with other memory instructions. The event's logical view is also recorded at the commit point of its operation, and is a set of events for all library operations that *happen-before* the operation in question. If an event e is in the logical view of another event d, *i.e.*, $e \in G(d).\mathtt{logview}$, we say that e happens before d. Technically, it is the commit instruction of e's operation that happens before the commit instruction of d's operation.

Intuitively, we use the logical view construction as an approximation of the hb relation between library operations, just as the physical view construction is an approximation of hb between memory instructions. The difference is that while physical views approximate hb *globally* between memory instructions, logical views only approximate hb *locally* for the library in question. As such, our logical views correspond to the *local happens-before* lhb relation of a library object introduced by Yacovet. Henceforth we use $e \in G(d).\mathtt{logview}$ and $(e, d) \in G.\mathsf{lhb}$ interchangeably.

The $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{hb}}$ style extends $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{so}}$ following a simple pattern: (1) the abstract state is accompanied by the graph that tracks all operations committed so far, and (2) at each operation's commit point, in addition to a potential update of the abstract state, a fresh event e representing the operation is added to the graph. For example, in Abs-Hb-Enq, when an enqueue of v commits, the current graph G of q is extended atomically with a fresh event e whose type is $\mathsf{Enq}(v)$, into G': $G \sqsubseteq G'$.

Local assertions for logical views. The partial orders are also extended at e's commit point to relate it to other operations. In Abs-Hb-Enq, $G'.\mathsf{lhb}$ extends $G.\mathsf{lhb}$ by setting $G'(e).\mathtt{logview} = M'$, the set containing all operations that happen before e. M' includes M_0—the *local logical view* of the calling thread, which tracks the operations that happen-before the **enq** call. This tracking of thread-local logical views is done by a new *persistent* assertion $\mathsf{SeenQueue}(q, G_0, M_0)$, where G_0 is a *snapshot* of the current G ($G_0 \sqsubseteq G$), and together with M_0 they accumulate (a lower bound on) the information about operations that the thread has synchronized with. For instance, after the call, the thread receives $\mathsf{SeenQueue}(q, G', M')$ with the latest snapshot G' and a new logical view M', reflecting that the thread has synchronized with more operations ($M_0 \sqsubseteq M'$), including the operation e that it has just executed ($e \in M'$). By taking SeenQueue as a local precondition, the specs can specify that the operation's behavior can depend on what has happened before it—we will shortly see how that allows us to use Abs-Hb-Deq to verify the MP client in Figure 1.

Compared to the $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{so}}$ style, in $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{hb}}$ each library type has a local logical view assertion like SeenQueue that plays a double role: (1) to track the thread-local logical view (as explained above) and also (2) to track persistent facts about the object like the $\mathsf{isQueue}(q)$ assertion in Abs-So-Enq. The logical view assertion plays the same role for logical views as the "seen view" assertion $\exists V$ does for physical views: the tracked current local view can be published into the "public domain" (*i.e.*, the shared graph for logical views, the shared location history or abstract state for physical views) so that it can be consumed by other threads.

Consistency conditions. The $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{hb}}$ style specifies properties of the abstract state and the partial orders through the library's *consistency conditions*. The consistency conditions

are invariant, *i.e.*, should be maintained by all operations, and are specific to each library type.

For example, an excerpt of QueueConsistent, the consistency conditions for the queue library type, is given at the bottom right of Figure 2. It requires, among other things, that enqueues and dequeues must follow the first-in-first-out principle (FIFO, Queue-FIFO), stated in a fashion that is not too strong for RMC (more about that below). The fact that QueueConsistent is maintained by all operations is encoded in Abs-Hb-Queue-Consistency: the queue ownership assertion $\mathsf{Queue}(q, vs, G)$, which is consumed and reproduced around the commit point, always implies consistency. So when Abs-Hb-Enq and Abs-Hb-Deq extend (vs, G) to new state (vs', G'), the operations can assume $\mathsf{QueueConsistent}(vs, G)$ and must then re-establish $\mathsf{QueueConsistent}(vs', G')$.

More specifically, if **deq** succeeds with a value v, Abs-Hb-Deq tells the client that $G'.\mathsf{so}$ extends $G.\mathsf{so}$ with a new pair (e, d) where d is the new successful event added by the dequeue operation and e is an existing enqueue event that d dequeues from. Therefore, through Abs-Hb-Queue-Consistency, the spec additionally says that (e, d) satisfies, among other things,[5] (1) Queue-Matches: the return value v of the dequeue d must match the value enqueued by e; and (2) Queue-FIFO: if there is another enqueue event e' that happens before e, then e' must already have been dequeued by some d' ($(e', d') \in G.\mathsf{so}$), and our d cannot happen before d' ($(d, d') \notin G.\mathsf{lhb}$). (The consistency conditions on enqueue events are elided, so we will not discuss them.)

Weaker but flexible. The Queue-FIFO condition appears weaker than what one might expect, *i.e.*, $(d', d) \in G.\mathsf{lhb}$, but such a condition only works for strongly synchronized (*e.g.*, SC) implementations. As stated, Queue-FIFO is also satisfiable by implementations that have little synchronization between dequeues. In fact, we have verified that Queue-FIFO is satisfiable by a fairly relaxed implementation (similar to the weak version in [62]) of the Herlihy-Wing queue [34]. The implementation ensures lhb only between matching enqueue-dequeue pairs, but not among enqueues or among dequeues. (As one might guess, enqueues use release operations, and dequeues use acquire ones.)

Nonetheless, Queue-FIFO is still flexible enough that, for example, if a client decides to use the queue in an SC fashion by adding sufficient external synchronization, the client can know that lhb is total, *i.e.*, $(d', d) \in G.\mathsf{lhb} \vee (d, d') \in G.\mathsf{lhb}$, and can thus exclude the right-hand side of the disjunction and regain the stronger FIFO condition with $(d', d) \in G.\mathsf{lhb}$. This demonstrates the benefits of more detailed partial orders: by specifying ordering between operations with more complex but seemingly weaker conditions, we can (1) require only minimal ordering from implementations, and at the same time (2) allow clients the flexibility to strengthen the specs by combining the library's exposed internal ordering with the client-generated external ordering.

Message-Passing client verification. When a call to **deq** returns empty (ϵ), consistency demands that the added empty dequeue event d satisfies Queue-EmpDeq, which is sufficient to verify the MP client (Figure 1). Intuitively, Queue-EmpDeq says that there cannot be another enqueue e which happens before d but has not been dequeued in G—if there were, then the dequeue would have successfully returned some element from the queue. The verification of MP depends on the fact that both enqueue events e_1 and e_2 done by the left-most thread, of which at most one can be consumed by the middle thread, happen before the dequeue of the right-most thread. By Queue-EmpDeq the dequeue cannot be an empty one and must dequeue from e_1 or e_2 and return either 41 or 42.

The proof sketch of this example in Compass is given in Figure 3. Following the pattern mentioned at the end of §2.2, we put the ownership $\mathsf{Queue}(q, \_)$ in an invariant to enforce a concurrent protocol on the queue, using a *dequeue permission* called deqPerm that can be defined with Iris ghost state [38]. One dequeue permission $\mathsf{deqPerm}(1)$ is needed to perform one successful dequeue. This requirement can be seen in the invariant: $\mathsf{deqPerm}(\mathsf{size}(G.\mathsf{so}))$ counts the number of successful dequeues, and a successful dequeue will extend $G.\mathsf{so}$ by 1, so anyone who successfully dequeues needs to put in a $\mathsf{deqPerm}(1)$ to re-establish the invariant. For our particular example, we also implement deqPerm such that there are only *two* $\mathsf{deqPerm}(1)$'s (*i.e.*, $\mathsf{deqPerm}(2)$) in the whole system. We then give one permission to each consumer thread before they run. Initially the queue is set to be empty, and all threads are given a persistent observation $\mathsf{SeenQueue}(q, \emptyset, \emptyset)$ of the initial empty state.

The verification of the left-most thread is straightforward: for each enqueue, we use LAInv-Acc to open the invariant and then use Abs-Hb-Enq. Afterwards the thread has two enqueue events $\{e_1, e_2\}$ in its logical view, and the write to flag releases $\mathsf{SeenQueue}(q, G_1, \{e_1, e_2\})$ to the right-most thread. The verification of the middle thread uses LAInv-Acc and Abs-Hb-Deq, and if the dequeue succeeds, $\mathsf{deqPerm}(1)$ can be given up to re-establish the client invariant. Finally, in the verification of the right-most thread, the acquire read of 1 from flag receives $\mathsf{SeenQueue}(q, G_1, \{e_1, e_2\})$ from the left-most thread. We then use LAInv-Acc and Abs-Hb-Deq to perform the dequeue, with $M_0 := \{e_1, e_2\}$. Before re-establishing the invariant, we inspect the resulting dequeue d_3. If it is a successful dequeue, we can put $\mathsf{deqPerm}(1)$ in the invariant and finish. If d_3 is an empty dequeue, we derive a contradiction. As there are only two $\mathsf{deqPerm}(1)$ permissions in the whole system, of which one is owned by the current (right-most) thread, when we open the invariant we know that the most up-to-date (right before d_3) graph G can have at most one dequeue: $\mathsf{size}(G.\mathsf{so}) \leq 1$. Furthermore, the thread has observed two enqueues, so in G there must be at least one

[5] For example, an element can only be dequeued once.

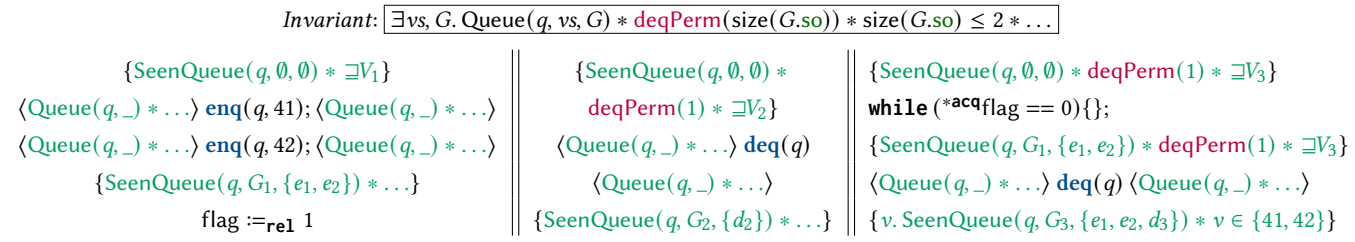

Figure 3. A proof sketch of Message Passing with queues.

enqueue that is not dequeued yet, which must be in $\{e_1, e_2\}$. Due to SeenQueue$(q, G_1, \{e_1, e_2\})$, both e_1 and e_2 happen before d_3. By Queue-EmpDeq, we have our contradiction. □

3.2 Weaker Specs by Abandoning Abstract States

The $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{hb}}$ specs are particularly strong and only satisfiable by strong implementations, because one must be able to construct the abstract state at commit points. For example, we have verified that a purely release-acquire implementation of the Michael-Scott queue [56] satisfies the $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{hb}}$ specs for queues (and therefore transitively the $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{so}}$ specs). The release-acquire memory model, though not as strong as the SC or Multicore OCaml model, still provides sufficient synchronization to construct the list of values *vs* in the queue.

However, it is extremely difficult to construct the abstract state for the relaxed Herlihy-Wing queue implementation mentioned above: it would require delicate reordering of commit points on the fly, and sometimes require *future-dependent* knowledge about dequeue operations. In fact, the verification of the LAT specs in the SC memory model for Herlihy-Wing queue relied on *prophecy variables* [39], whose application in RMC is still an open research problem. In this work we instead verify the relaxed Herlihy-Wing implementation against $\mathsf{LAT}_{\mathsf{hb}}$ specs, a weakening of the $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{hb}}$ specs where the abstract state is abandoned. In particular, our instance of the $\mathsf{LAT}_{\mathsf{hb}}$ specs for queues is exactly the specs Abs-Hb-Enq and Abs-Hb-Deq (Figure 2) *without* *vs*.

$\mathsf{LAT}_{\mathsf{hb}}$ specs may appear weak, but they can still take advantage of external synchronization information, *i.e.*, the argument in §3.1 about flexibility of the partial orders still applies. Practically, they are sufficient to verify the MP client in Figure 1. We can also use them to verify the following *single-producer single-consumer* (SPSC) client of a queue:

$$\begin{array}{c}\left\{\begin{array}{l}\mathsf{SeenQueue}(q, \_, \_) * \\ a_p \mapsto [a_0, \ldots, a_{n-1}] * \ldots\end{array}\right\} \\ \textbf{produce}(q, a_p, 0, n) \\ \{a_p \mapsto [a_0, \ldots, a_{n-1}] * \ldots\}\end{array} \;\Big\|\; \begin{array}{c}\left\{\begin{array}{l}\mathsf{SeenQueue}(q, \_, \_) * \\ a_c \mapsto [0, \ldots, 0] * \ldots\end{array}\right\} \\ \textbf{consume}(q, a_c, 0, n) \\ \{a_c \mapsto [a_0, \ldots, a_{n-1}] * \ldots\}\end{array}$$

Here, there is only one thread performing enqueues—the **produce**r—and only one thread performing dequeues—the **consume**r. The **produce**r reads the array a_p for elements with the indices in $[0, n)$ and enqueues them in that order, while the **consume**r keeps dequeueing for n elements and writes them in the indices $[0, n)$ of the array a_c in the dequeueing order. The expected behavior is FIFO: in the end the array a_c should have the same elements as a_p.

To verify this example, we use the $\mathsf{LAT}_{\mathsf{hb}}$ specs for queues (*i.e.*, Abs-Hb-Enq and Abs-Hb-Deq *without* abstract states) to derive the *stronger* $\mathsf{LAT}_{\mathsf{hb}}$-style specs for SPSC queues [18], simply by building a concurrent SPSC client protocol. In this derivation, thanks to logical atomicity, at every commit point of a successful dequeue we can easily match it up with the right enqueue and thus prove FIFO. With the SPSC $\mathsf{LAT}_{\mathsf{hb}}$ specs, the example's verification is straightforward.

3.3 Stronger Specs with a Linearization

One may instead wish to specify stronger implementations more tightly with stronger specs. For example, Yacovet proposes strong specs where a library's operations are *linearizable* but with weaker synchronization requirements. We call the encoding of these specs in Compass with logical atomicity the $\mathsf{LAT}^{\mathsf{hist}}_{\mathsf{hb}}$ style, and it is a strengthening of the $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{hb}}$ specs with a *linearizable history* H.[6] An excerpt of the instance for *stacks* is given in Figure 4. The linearizable history H subsumes both the event graph G and the abstract state *vs*: it not only tracks the partial orders between operations but additionally gives them a *total order* to that can be interpreted to compute the abstract state.

to is a *linearization* of the operations and can be considered as a sequential specification. However, to has weaker synchronization requirements than traditional linearizability because it does not *imply* lhb, but only needs to *respect* lhb (*i.e.*, $H.\mathsf{lhb} \subseteq \mathsf{to}$). Additionally, to directly encodes the stack's LIFO property, as well as stricter behaviors of failing empty pop operations, as required by Hist-Hb-Stack-Linearizable. That is, to is a *reordering* (permutation) of H's operations, satisfying interp(to, *vs*) for some abstract state *vs*, through which we can look at a concurrent stack's history as if it were the history of a sequential stack, in the same fashion as in classical linearizability: a successful push adds a new element to the stack's head, a successful pop removes and returns the head element, and an empty pop only happens

[6] Note that the concept for histories of library operations is different from that for histories of write events to locations in §2.3.

Hist-Hb-Push
$\text{SeenStack}(s, H_0, M_0) * \sqsupseteq V \vdash$
$\langle H.\ \text{Stack}(s, H)\rangle\ \textbf{push}(s, v)\ \langle().\ \text{Stack}(s, H +\!\!+ [(\text{Push}(v), \ldots)])\ast\ldots\rangle$

Hist-Hb-Pop
$\text{SeenStack}(s, H_0, M_0) * \sqsupseteq V \vdash$
$\langle H.\ \text{Stack}(s, H)\rangle\ \textbf{pop}(s)\ \langle v.\ \text{Stack}(s, H +\!\!+ [(\text{Pop}(v), \ldots)])\ast\ldots\rangle$

Hist-Hb-Stack-Linearizable
$\text{Stack}(s, H) \vdash$
$\exists \text{to}, vs.\ H.\text{lhb} \subseteq \text{to} \wedge \text{to} = \text{permute}(H) \wedge \text{interp}(\text{to}, vs) \wedge \ldots$

$\text{interp}(\text{to}, vs) ::=$
$\quad \text{to} = [\,] \vee (\exists e, \text{to}', vs'.\ \text{to} = \text{to}' +\!\!+ [e] \wedge \text{interp}(\text{to}', vs')$
$\quad\quad \wedge ((e = \text{Push}(v) \wedge vs = (v, \_) :: vs')$
$\quad\quad\quad \vee (e = \text{Pop}(v) \wedge vs' = (v, \_) :: vs)$
$\quad\quad\quad \vee (e = \text{Pop}(\epsilon) \wedge vs = vs' = [\,]))$

Figure 4. $\text{LAT}^{\text{hist}}_{\text{hb}}$ specs for "linearizable" stacks (excerpt).

if the stack is truly empty. Note, however, that this is only a perspective on histories—at the *commit point* of an empty pop, the spec does *not* say that the stack is necessarily empty.

Beyond local-happens-before with logical views. We have verified that the $\text{LAT}^{\text{hist}}_{\text{hb}}$ specs are satisfied by a relaxed implementation of the Treiber stack [70]. One of the main verification challenges is the construction of to. We wish to construct to from the higher-level abstraction of the partial orders on the stack's operations, and avoid having to perform RMC reasoning directly about memory locations in the process. The exposed lhb relation, however, is not sufficient to construct to because our RMC Treiber stack implementation is fairly relaxed: push operations use release CASes and successful pop operations use acquire CASes, and thus there are only lhb edges between matching push-pop pairs. Looking at the implementation, though, such a to *is* derivable from the ordering of the CAS instructions done to the stack's head pointer by successful push and pop operations. Fortunately, we can repurpose our logical view setup to expose not just lhb, but the stronger partial order that includes both lhb *and* the modification order on the stack's head created by CASes. We can then construct a to from this stronger order, without having to deal with lower-level RMC reasoning.

Until now we have equated partial orders with lhb, but here we see that this need not be the case, and that by using logical views to expose a richer partial order we can verify a fairly relaxed implementation against reasonably strong, linearizability-style $\text{LAT}^{\text{hist}}_{\text{hb}}$ specs.

4 Compositional Verification of the Elimination Stack

In this section, we briefly demonstrate the application of our specs to verify an RMC implementation of the elimination stack [32]. This verification is both a *client* verification and a *library* verification: the elimination stack is a client that composes an underlying base stack and an exchanger.

4.1 The Elimination Stack

The idea for the elimination stack (ES) comes from a simple observation: if a **push** is immediately followed by a **pop**, then the stack appears unchanged, and that **push** and **pop** are said to *eliminate* each other. The elimination mechanism can be implemented with an exchanger (which in turn can be implemented as an array of exchangers) that supports concurrent exchanges of data with arbitrary matching. A thread simply calls **exchange**(x, v_1) on the exchanger object x with some value $v_1 \neq \bot$. If the return value is $\bot$, the exchange has failed, but if it is some $v_2 \neq \bot$, then the thread has successfully exchanged v_1 for v_2 with another thread. Additionally, the two threads *synchronize with each other*,[7] which from the separation logic perspective supports *resource exchanges* between the matching threads.

The ES *try* operations, which can fail due to contention, can be implemented simply by composing the two libraries without any extra synchronization, as follows:

```
try_push(s, v) ::= if try_push'(s.base, v) then true
                   else exchange(s.ex, v) == SENTINEL
try_pop(s) ::= let v = try_pop'(s.base) in
               if v != FAIL_RACE then v else
               let v' = exchange(s.ex, SENTINEL) in
               if v' ∉ {SENTINEL, ⊥} then v'
               else FAIL_RACE
```

Each operation first tries the base stack's corresponding operation, and if that fails due to contention, it tries to use the exchanger to match another operation without going through the base stack. More specifically, **try_push**(s, v) calls the base stack's own **try_push′** and returns **true** (signifying success) if that succeeds. Otherwise, it calls **exchange** (on s.ex) and returns **true** only if its exchange is successfully matched with a pop operation, signified by the SENTINEL value. Similarly, **try_pop**(s) calls the base stack's **try_pop′** and returns v only if **try_pop′** did not fail due to contention (FAIL_RACE). (**try_pop** returns empty ϵ if **try_pop′** does.) Otherwise, **try_pop** calls **exchange** with SENTINEL, and only succeeds with the returned value v' if it is matched ($v' \neq \bot$) with a push ($v' \neq$ SENTINEL).

Verification results. Assuming the LAT_{hb}-style specs for the base stack and the exchanger (Figure 5), we have verified that our relaxed ES implementation satisfies the same LAT_{hb}

[7] Technically, the two commit points of the matching exchanges are not both in hb with each other—it is counterintuitive to have cycles in hb—but it is the case that the *beginning* of one **exchange** call happens before the *end* of its matching **exchange** call. We needed to extend our specs to account for this subtlety, but, due to space constraints, we elide it from the specs and from the discussion here.

specs as the base stack. The $\mathsf{LAT}_{\mathsf{hb}}$ specs for stacks are very similar to those for queues—the key difference is the change from FIFO to LIFO in consistency. (Please see the full specs in [18].) Since we have verified, *separately*, that our RMC Treiber stack and exchanger implementations satisfy their $\mathsf{LAT}_{\mathsf{hb}}$ specs, we easily get a closed-proof verification of an ES built from those two implementations. We assumed the $\mathsf{LAT}_{\mathsf{hb}}$ specs for the base stack to demonstrate that the verification does not rely on very strong properties, but, since the proofs are modular enough, we conjecture that the same proofs can be applied (with minor modifications) to show that, if the base stack satisfies the stronger $\mathsf{LAT}^{\mathsf{hist}}_{\mathsf{hb}}$ or $\mathsf{LAT}^{\mathsf{abs}}_{\mathsf{hb}}$ specs, then the ES implementation also satisfies the same stronger specs.

Compositional verification. The ES verification does not involve RMC reasoning, because the implementation does not add any new atomic instructions. The core work is composing the base stack's events and the exchanger's events into the ES events in a way that satisfies the consistency conditions for stacks. This can be seen as a simulation proof with a simple simulation relation: every base stack operation is simulated by a corresponding ES operation, and successful matching exchange pairs between a non-SENTINEL value and a SENTINEL value are simulated by an ES push and an ES pop respectively. (Other exchange events are ignored by the simulation.)

The non-trivial parts of the proof are where the simulation needs to simulate commit points and maintain consistency: whenever a base operation commits, the ES operation needs to commit accordingly, and needs to re-establish the ES consistency conditions using the consistency conditions of the base stack and the exchanger. The re-establishment of consistency relies crucially on the fact that eliminations are *atomic*: the commits of ES push and pop events that originate from a pair of matching exchanges need to be performed together at once, so that the pushed element is popped immediately, and no (commit points of) other concurrent ES operations can observe the intermediate state where the ES push has already been committed but the ES pop has not. This atomicity property of the exchange-based ES event pairs is crucially needed for LIFO. We discuss how this property shows up in the exchanger specs next.

4.2 Strong Specs for Exchangers

A simplified $\mathsf{LAT}_{\mathsf{hb}}$-style spec for the **exchange** function is shown in Hb-Exchange (Figure 5). The spec involves a local logical view assertion $\mathsf{SeenExchanges}(x, G_0, M_0)$, and an atomically shared ownership assertion $\mathsf{Exchanger}(x, G)$ for the exchanger object x. At the commit point, the current graph G is extended with a new event e_1 with type $\mathsf{Exchange}(v_1, v_2)$, where v_2 is the returned value. If the exchange fails, the return value v_2 is $\bot$ and the event type is $\mathsf{Exchange}(v_1, \bot)$. If the exchange succeeds, it can only succeed together with another exchange identified by e_2, and the $G.\mathsf{so}$ relation is extended with the two events in both directions ($\{(e_1, e_2), (e_2, e_1)\}$), signifying that they are synchronized with each other.

The remaining part of the spec is to maintain the perspective that *a matching pair of exchanges is committed atomically together*: it is important that there can be *no interference* between the two commits of the matching exchanges. In other words, no other thread should be able to observe an incomplete state of the exchanger where one successful exchange has been committed but its matching exchange has not. But how can *two* commit points be atomic? This conflicts with the intuitive interpretation of LATs that there exists a committing instruction c within each logical operation! To resolve this conundrum, we need *helping*.

Helping for atomicity. Helping is a pattern where one operation—the helper—helps to perform the commit (the update to the shared state) of another operation—the helpee. This means that the commit point of the helpee is not within its own execution, but rather within the helper's execution. For the matching exchange pairs, the commit points coincide: at the helper exchange's commit point, it atomically performs the helpee exchange's commit and then its own commit. This is materialized in the successful case of Hb-Exchange with (1) a *commit order* ($<$) of the events and (2) the addition of a *local postcondition* (in red, $\{\ldots\}$) that only holds once the function returns (rather than at the commit point).

The commit order $<$ on events is the logical order in which the events are committed to the shared graph G. In Hb-Exchange, the commit order between a matching exchange pair dictates who the helper is, and how each commit updates the shared graph G. If the *current* exchange e_1 is committed before the *other* exchange e_2, *i.e.*, $e_1 < e_2$, then e_2 is the helper. Otherwise, if $e_2 < e_1$, then e_1 is the helper.

Since the helper atomically performs the helpee's update and then its own update, it always knows the result of the helpee's update, while the helpee will only learn about the helper's update *after* both commits have been completed. This is the asymmetry in Hb-Exchange: if e_1 is the helpee, it only adds itself to the current graph G: $G' = G[e_1 \mapsto (\mathsf{Exchange}(v_1, v_2), V_1, M')]$; but if e_1 is the helper, it knows that the helpee's event e_2 must already be in the current graph: $G(e_2) = (\mathsf{Exchange}(v_2, v_1), V_2, M')$, and the helper not only adds itself to the current graph G, but also extends $G.\mathsf{so}$ with the pairs $\{(e_1, e_2), (e_2, e_1)\}$. The client thread of the helper learns all of this information about the updated G' atomically right after the helper's commit, by which point it has also locally observed both e_1 and e_2, via $\mathsf{SeenExchanges}(x, G', M')$ and $\{e_1, e_2\} \subseteq M'$. The client thread of the helpee, on the other hand, right after its own commit has only locally observed its own event e_1, via $\mathsf{SeenExchanges}(x, G', M' \setminus \{e_2\})$, because the helper commit has not been performed yet and e_2 has not been added to G'. Only in the local postcondition (in $\{\ldots\}$), after both

HB-PUSH

$$\mathsf{SeenStack}(s, G_0, M_0) * \sqsupseteq V \vdash \langle G.\, \mathsf{Stack}(s, G)\rangle\ \mathbf{push}(s, v)\ \left\langle (). \exists G' \sqsupseteq G, M' \supseteq M_0, V' \sqsupseteq V.\, \mathsf{Stack}(s, G') * \mathsf{SeenStack}(s, G', M') * \sqsupseteq V' \right.$$
$$\left. * \exists e \notin G.\, e \in M' \wedge G' = G\big[e \mapsto (\mathsf{Push}(v), V', M')\big] \right\rangle$$

HB-POP

$$\mathsf{SeenStack}(s, G_0, M_0) * \sqsupseteq V \vdash \langle G.\, \mathsf{Stack}(s, G)\rangle\ \mathbf{pop}(s)\ \left\langle v. \exists G' \sqsupseteq G, M' \supseteq M_0, V' \sqsupseteq V.\, \mathsf{Stack}(s, G') * \mathsf{SeenStack}(s, G', M') * \sqsupseteq V' \right.$$
$$\left. * \big(\ldots \wedge G' = G[d \mapsto (\mathsf{Pop}(\epsilon), \ldots)]\big) \vee \big(\ldots \wedge G' = G[d \mapsto (\mathsf{Pop}(v), \ldots)] \wedge \ldots\big) \right\rangle$$

HB-EXCHANGE

$$\mathsf{SeenExchanges}(x, G_0, M_0) * v_1 \neq \bot * \sqsupseteq V_0 \vdash \langle b, G.\, \mathsf{Exchanger}(x, G)\rangle\ \mathbf{exchange}(x, v_1)$$

$$\left\langle v_2. \exists G' \sqsupseteq G, M' \supseteq M_0, V_1 \sqsupseteq V_0, V_2, e_1 \notin G, e_2.\, \mathsf{Exchanger}(x, G') * \sqsupseteq V_1 * e_1 \in M' * G' = G\big[e_1 \mapsto (\mathsf{Exchange}(v_1, v_2), V_1, M')\big] \right.$$
$$\left. * \vee \begin{cases} v_2 = \bot & * \mathsf{SeenExchanges}(x, G', M') * \mathsf{ExchangerConsistent}(G') \\ v_2 \neq \bot \wedge e_1 < e_2 & * e_2 \in M' * \mathsf{SeenExchanges}(x, G', M' \setminus \{e_2\}) \\ v_2 \neq \bot \wedge e_2 < e_1 & * e_2 \in M' * \mathsf{SeenExchanges}(x, G', M') * \mathsf{ExchangerConsistent}(G') \\ & \quad * G(e_2) = (\mathsf{Exchange}(v_2, v_1), V_2, M') * G'.\mathsf{so} = \{(e_1, e_2), (e_2, e_1)\} \cup G.\mathsf{so} \end{cases} \right\rangle$$

$$\left\{ (v_2 \neq \bot \wedge e_1 < e_2) \Rightarrow \exists G'' \sqsupseteq G'.\, \mathsf{SeenExchanges}(x, G'', M') * \mathsf{ExchangerConsistent}(G'') \right.$$
$$\left. * G''(e_2) = (\mathsf{Exchange}(v_2, v_1), V_2, M') * G''.\mathsf{so} = \{(e_1, e_2), (e_2, e_1)\} \cup G'.\mathsf{so} \right\}$$

Figure 5. $\mathsf{LAT}_{\mathsf{hb}}$-style specs for stacks and exchangers (excerpt, simplified).

commits have been performed, can the helpee learn about the new graph G'' (e_2's G') that completes e_1's G' (e_2's G) with e_2 and the so pairs, and locally observe both events, via $\mathsf{SeenExchanges}(x, G'', M')$.

Intermediate states. That matching exchange pairs are committed atomically together is also reflected by the fact that we do not always have consistency: the ownership $\mathsf{Exchanger}(x, G)$ does *not* imply $\mathsf{ExchangerConsistent}(G)$. Instead, we have $\mathsf{ExchangerConsistent}(G')$ only with a completed graph G', *i.e.*, after the failure case or after the helper's commit. Between the helpee's commit and the helper's commit, the exchanger is in an incomplete intermediate state.

As such, those intermediate states can appear in a client invariant. However, it is important that the client needs to handle such states *only when* it uses the exchanger, and that other non-exchanger-related operations will never observe those states. For example, the invariant of the elimination stack needs to consider the intermediate state where a push event created by a successful exchange is inserted into the graph, but the matching pop event by the matching exchange is not. A successful push using the base stack and running concurrently with the exchange pair should *not* observe the client invariant in such an intermediate state, because it would not be able to prove LIFO then.

Our full exchanger spec (see our supplementary materials [18]) supports this form of intermediate state reasoning: when using the exchanger, the client need not maintain its invariant for the intermediate state between the two commits; it only needs to re-establish its invariant after both commits. When *not* using the exchanger, the client invariant is never in such intermediate states.

Strength of the specs. To the best of our knowledge, the full exchanger spec is the first ever proposed CSL spec for RMC exchangers. It is strong enough for the proof of the elimination stack (§4.1), and we have also used it to derive a spec that supports resource exchanges, where each **exchange** call needs to provide the resources to be exchanged only at its commit point, and only if the exchange succeeds.

5 Extensions to iRC11

In this section, we briefly explain the main extensions to iRC11 needed to state and prove useful logically atomic COMPASS specs. These extensions include *objective invariants*, *view-explicit modalities*, and *atomic points-to assertions*. Our supplementary materials and Coq development [18] provide more details on these extensions.

5.1 Objective Invariants

Recall the rule LAINV-ACC in §2.2 which relates LATs and invariants: a logically atomic expression can access invariants around its commit point. The rule is sound in SC, but is problematic in the logic of iRC11, as general invariants that can contain arbitrary resources do not exist in iRC11. Intuitively, when moving resources owned by a thread—which are interpreted according to that thread's local views—into the "public domain" of an invariant, we have to pick the views used to interpret those resources, now that they are no longer tied to a thread. Following RSL, FSL, and GPS [22, 23, 41, 72, 73], iRC11 sets the view for a shared resource to be a view of some location's write event, effectively restricting invariants to *single locations*. That is, iRC11's version of invariants cannot contain arbitrary resources, but only ownership of a single

location and resources associated with its accesses. Unfortunately, abstract ownership of a data structure typically consists of multiple locations. For example, Queue(q, G) should include ownership of all of the queue's constituent memory locations—its head and tail pointers, as well as its elements. Single-location invariants thus are insufficient for Compass.

Another solution to the view conundrum is to require that resources put inside invariants are always *objective*, in the sense that they hold at *any* view, *i.e.*, if I holds at some view V_1, then it also holds at some other view V_2. As such, we can pick any view to interpret objective resources when moving them from a thread into an invariant. This gives rise to *objective invariants*, an experimental, unpublished construct of iRC11 that has proved useful in Cosmo to achieve strong LAT specs. For Compass, we adopt objective invariants and develop a complete, official interface for them in iRC11.

Objective invariants are sound for RMC, and yet have almost the same interface as SC invariants. In fact, they admit both Inv-Acc and LAInv-Acc, so they can be combined with LATs to achieve both strong functional correctness and modular client reasoning, in the way that we have explained in §2 and §3. The only difference between objective invariants and SC-logic invariants is in the invariant allocation rule.

$$\textsc{OInv-alloc}\ \frac{\mathsf{objective}(\mathcal{I})}{\triangleright \mathcal{I} \Rrightarrow \boxed{\mathcal{I}}}$$

That is, if we have $\triangleright \mathcal{I}$, we can put it in an invariant as long as we can show that it is objective, *i.e.*, objective$(\mathcal{I})$. The objective side-condition is where we are obliged to take care of the relaxed memory effects. To make our Compass specs compatible with objective invariants, we additionally require that the abstract ownership of a data structure—*e.g.*, Queue(q, G) for queues or Stack(s, H) for stacks—is objective.

But, how do we make sure that the abstract ownership of a queue is objective? While many resources such as pure facts and ghost ownership are *view-independent* and thus objective, most resources, including the usual points-to assertion, are not, because their interpretations depend on the view observations of the owner. In order to make them objective or compatible with objectivity, we need the ability to *briefly* perform explicit view reasoning—which is hidden by the logic of iRC11—with the help of view-explicit modalities and atomic points-to assertions.

5.2 View-Explicit Modalities

We have seen one such view-explicit modality in §2.3: the seen-view assertion $\sqsupseteq V$ says that its owner's observations are at least V. Another important modality is the *view-at* modality $@_V P$ which asserts that P holds explicitly (at least) at the view V. This means that the interpretation of P is now justified by the view V and not by the owning thread's local view. The view-at modality has the following rules:

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

$@_V P$ is objective, because it no longer depends on the view of the owner. The introduction rule VA-intro allows us to *freeze* an owned resource P at some view V that we have observed ($\sqsupseteq V$). Having done this, we can send $@_V P$ and $\sqsupseteq V$ away on different routes: $@_V P$ can be put inside an invariant, and $\sqsupseteq V$ can be passed to another thread using atomic operations—recall Rel-Write and Acq-Read in §2.3. Anyone who receives both parts can use VA-elim to regain P.

Consequently, the view-at modality allows us to turn arbitrary resources into objective assertions with an explicit view V, and move them into an objective invariant. The invariant then needs to track these views carefully, *e.g.*, by relating them to the views of some location's writes. Then when another thread interacts with the invariant, it can relate those views to its own seen-view assertions ($\sqsupseteq V$), eliminate the view-at modality, and locally acquire the resources.

5.3 Atomic Points-To Assertion

To complete the story, we need stronger rules than Rel-Write and Acq-Read that work with the atomic points-to assertions stored under a view-at modality.

AT-Rel-Write

$$\left\{\sqsupseteq V * \ell \sqsupseteq_{\mathsf{sn}} h_0 * @_{V_b} \ell \mapsto_{\mathsf{at}} h\right\}\ \ell :=_{\mathtt{rel}} v$$
$$\left\{\begin{array}{l}(). \exists t \notin h \supseteq h_0, V'. \mathsf{max}(\{V(\ell)\} \cup \mathrm{dom}(h_0)) < t * \\ \quad V \sqcup \{\ell \mapsto t\} \sqsubseteq V' * \sqsupseteq V' * \\ \quad \ell \sqsupseteq_{\mathsf{sn}} h_0[t \mapsto (v, V')] * @_{V_b \sqcup V'}(\ell \mapsto_{\mathsf{at}} h[t \mapsto (v, V')])\end{array}\right\}$$

AT-Acq-Read

$$\left\{\sqsupseteq V * \ell \sqsupseteq_{\mathsf{sn}} h_0 * @_{V_b} \ell \mapsto_{\mathsf{at}} h\right\}\ {}^{*\mathtt{acq}}\ell$$
$$\left\{\begin{array}{l}v. \exists t \geq V(\ell), V_t, h', V' \sqsupseteq V_t. h_0 \subseteq h' \subseteq h * h'(t) = (v, V_t) \\ \quad * \mathsf{max}(\mathrm{dom}(h_0)) \leq t * \sqsupseteq V' * \ell \sqsupseteq_{\mathsf{sn}} h' * @_{V_b \sqcup V'} \ell \mapsto_{\mathsf{at}} h\end{array}\right\}$$

In these rules, we do not require the ownership of atomic points-to $\ell \mapsto_{\mathsf{at}} h$ locally. Instead, we need it only objectively, under a view-at modality at some view V_b, but together with some *local history observation* $\ell \sqsupseteq_{\mathsf{sn}} h_0$ stating that the calling thread has observed a snapshot h_0 of the full history h ($h_0 \subseteq h$). The history observation ensures that the thread has made some basic observations about the location (*e.g.*, that it has been initialized). After the access, the rules return the atomic points-to, still under the view-at modality, at the view $V_b \sqcup V'$ where V' is the calling thread's new local view.

These stronger rules are compatible with atomic points-to ownership stored inside an objective invariant, under a view-at modality. For example, in our verification of the Michael-Scott queue against the $\mathsf{LAT}_{\mathsf{hb}}$ specs for queues, we can define the queue's abstract ownership Queue(q, G) as ownership of the head and tail pointers as well as the queue elements, all under a view-at modality at some existentially

quantified view V_b. This makes Queue(q, G) an objective resource that can be shared concurrently, and yet Queue(q, G) is still sufficient to allow atomic accesses on q's memory locations. We therefore achieve the benefits of logical atomicity even in the presence of relaxed memory effects.

As a final note, the view-explicit modalities and the atomic points-to also exist and play a key role in Cosmo, albeit in a much simpler form. The interface for our view-explicit modalities and atomic points-to is much more extensive, in order to support the various access modes of the weaker ORC11 memory model. These details can be found in the supplementary materials accompanying this paper [18].

6 Related and Future Work

Our specification styles build on extensive prior work in relaxed correctness conditions, and in program logics for fine-grained concurrent SC and RMC programs.

Relaxed correctness conditions. Various alternative correctness conditions to linearizability have been developed [20, 29, 33, 36, 58], particularly for distributed systems [3, 9, 10, 57] and relaxed memory [4, 8, 21, 25, 26, 35, 45, 62]. Most of these were developed outside a program logic, directly on complex low-level concurrency semantics, and with little support for client reasoning or mechanization. As discussed in §1.2, we believe the Yacovet approach [62] is the most general of these. By enhancing Yacovet specs in Compass with logical atomicity, we demonstrate that existing relaxed correctness conditions can be used in combination with separation logic to achieve stronger and better modular client reasoning as well as more foundational (mechanized) verifications. We consider it future work to encode more of these relaxed correctness conditions in Compass.

SC program logics. Logical atomicity is just one CSL alternative to linearizability. Another is to avoid identifying commit points and instead reason directly about *refinements* between a sequential "specification" program and the concurrent implementation program [28, 46, 52, 71]. However, sequential specs are not always suitable as correctness conditions (*e.g.*, for exchangers), and *non-sequential* refinement is still an open problem for RMC logics. Our work demonstrates the usefulness of logical atomicity in RMC. As future work, we consider adapting *prophecy variables* [1, 2, 39] to our framework, as they may help simplify our specs.

FCSL [19, 64, 65] and the rely-guarantee-based Hoare logics by Hemed et al. [31] and Khyzha et al. [43] support specifying non-linearizable SC data structures with histories by encoding histories as auxiliary (ghost) state and by exposing partial, subjective views of the histories to clients. This is similar to our construction of graphs or linear histories. Compass can be seen as extending these logics with logical atomicity and RMC.

RMC program logics. Dalvandi and Dongol [15, 16], in parallel work, try to achieve the same goal of providing compositional specs and modular client reasoning for RMC data structures. Their approach uses an Owicki-Gries-style Hoare logic [14] for a more limited fragment of RC11 called RAR (which only has release-acquire and relaxed accesses, not non-atomics or fences). They specify libraries with view-based, atomic *abstract object semantics* for the library's operations, treating the object methods as primitives of the language. Client verifications rely on Hoare-triple specs derived directly from the abstract object semantics. To verify an implementation against a spec, they prove refinement, showing that synchronizations (in the view semantics) of the abstract object are simulated by synchronizations in the implementation. Their approach therefore shares similar ideas to ours (their specs are most similar to our $\mathsf{LAT}^{\mathsf{hist}}_{\mathsf{hb}}$ specs). The main limitation is in their simulation method: it applies only to *synchronization-free* clients, *i.e.*, those who synchronize only through the library in question. This is because it is nontrivial to characterize how external synchronizations affect the simulation relation. Consequently, they cannot obtain an end-to-end proof for clients that use external synchronizations, *e.g.*, the MP client in Figure 1. Furthermore, the use of Owicki-Gries-style logic means that they have to deal with additional *interference freedom* proofs. They report only one mechanized library verification, for the Treiber stack, with 12KLOC in Isabelle. In comparison, our mechanization results are more extensive, and our Treiber stack verification takes only 2.2KLOC in Iris, in Coq.

Several CSLs for RMC have been developed within Iris [17, 41, 54]. Our logic extends iRC11 and follows Cosmo in exposing more view information in specs. Our key innovation is the use of logical views on library operations, allowing us to give stronger specifications that can describe interactions with external synchronization. In retrospect, we believe that views are a concise, compositional, and user-friendly tool to describe the different kinds of synchronization that may occur in and around a data structure, and thus are useful for formulating full functional correctness specs under RMC.

Finally, as future work, we would like to apply the Compass approach to more sophisticated RMC libraries such as work-stealing queues [12, 50] and safe memory reclamation schemes for lock-free data structures [27, 55].

Acknowledgments

We would like to thank Azalea Raad for helpful conversations. This research was supported, in part, by European Research Council (ERC) Consolidator Grants for the projects "RustBelt" and "PERSIST", funded under the European Union's Horizon 2020 Framework Programme (grant agreements no. 683289 and 101003349, respectively), by a National Research Foundation of Korea (NRF) grant, funded by the Korea government (MSIT) (grant no. 2020R1C1C1010015), and by a (US) National Science Foundation grant (grant no. 1811894).

References

[1] Martín Abadi and Leslie Lamport. 1988. The Existence of Refinement Mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. IEEE Computer Society, 165–175. https://doi.org/10.1109/LICS.1988.5115

[2] Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

[3] Yehuda Afek, Guy Korland, and Eitan Yanovsky. 2010. Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. In *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6490)*. Springer, 395–410. https://doi.org/10.1007/978-3-642-17653-1_29

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

[5] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 55–66. https://doi.org/10.1145/1926385.1926394

[6] John Bender and Jens Palsberg. 2019. A formalization of Java's concurrent access modes. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 142:1–142:28. https://doi.org/10.1145/3360568

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

[8] Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. 2012. Concurrent Library Correctness on the TSO Memory Model. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 87–107. https://doi.org/10.1007/978-3-642-28869-2_5

[9] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated data types: specification, verification, optimality. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. ACM, 271–284. https://doi.org/10.1145/2535838.2535848

[10] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. 2015. Specifying Concurrent Problems: Beyond Linearizability and up to Tasks - (Extended Abstract). In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9363)*. Springer, 420–435. https://doi.org/10.1007/978-3-662-48653-5_28

[11] Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.* 3, POPL (2019), 70:1–70:28. https://doi.org/10.1145/3290383

[12] David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*. ACM, 21–28. https://doi.org/10.1145/1073970.1073974

[13] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*. Springer, 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

[14] Sadegh Dalvandi, Simon Doherty, Brijesh Dongol, and Heike Wehrheim. 2020. Owicki-Gries Reasoning for C11 RAR. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.11

[15] Sadegh Dalvandi and Brijesh Dongol. 2021. Verifying C11-style weak memory libraries. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*. ACM, 451–453. https://doi.org/10.1145/3437801.3441619

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

[17] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *Proc. ACM Program. Lang.* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

[18] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2021. Accompanying supplementary materials and Coq development of Compass. Available at https://plv.mpi-sws.org/compass/.

[19] Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2017. Concurrent Data Structures Linked in Time. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:30. https://doi.org/10.4230/LIPIcs.ECOOP.2017.8

[20] John Derrick, Brijesh Dongol, Gerhard Schellhorn, Bogdan Tofan, Oleg Travkin, and Heike Wehrheim. 2014. Quiescent Consistency: Defining and Verifying Relaxed Linearizability. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8442)*. Springer, 200–214. https://doi.org/10.1007/978-3-319-06410-9_15

[21] Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. 2018. Making Linearizability Compositional for Partially Ordered Executions. In *Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11023)*. Springer, 110–129. https://doi.org/10.1007/978-3-319-98938-9_7

[22] Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*. Springer, 413–430. https://doi.org/10.1007/978-3-662-49122-5_20

[23] Marko Doko and Viktor Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*. Springer, 448–475. https://doi.org/10.1007/978-3-662-54434-1_17

[24] Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding data races in space and time. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, 242–255. https://doi.org/10.1145/3192366.3192421

[25] Brijesh Dongol, Radha Jagadeesan, James Riely, and Alasdair Armstrong. 2018. On abstraction and compositionality for weak-memory linearisability. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10747)*. Springer, 183–204. https://doi.org/10.1007/978-3-319-73721-8_9

[26] Michael Emmi and Constantin Enea. 2019. Weak-consistency specification via visibility relaxation. *Proc. ACM Program. Lang.* 3, POPL (2019), 60:1–60:28. https://doi.org/10.1145/3290373

[27] Keir Fraser. 2004. *Practical lock-freedom.* Ph. D. Dissertation. University of Cambridge.

[28] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018.* ACM, 442–451. https://doi.org/10.1145/3209108.3209174

[29] Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. 2016. Local Linearizability for Concurrent Container-Type Data Structures. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada (LIPIcs, Vol. 59).* Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:15. https://doi.org/10.4230/LIPIcs.CONCUR.2016.6

[30] Mengda He, Viktor Vafeiadis, Shengchao Qin, and João F. Ferreira. 2018. GPS+: Reasoning About Fences and Relaxed Atomics. *Int. J. Parallel Program.* 46, 6 (2018), 1157–1183. https://doi.org/10.1007/s10766-017-0518-x

[31] Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. 2015. Modular Verification of Concurrency-Aware Linearizability. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9363).* Springer, 371–387. https://doi.org/10.1007/978-3-662-48653-5_25

[32] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A scalable lock-free stack algorithm. In *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain.* ACM, 206–215. https://doi.org/10.1145/1007912.1007944

[33] Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. 2013. Quantitative relaxation of concurrent data structures. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013.* ACM, 317–328. https://doi.org/10.1145/2429069.2429109

[34] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[35] Radha Jagadeesan, Gustavo Petri, Corin Pitcher, and James Riely. 2013. Quarantining Weakness - Compositional Reasoning under Relaxed Memory Models (Extended Abstract). In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792).* Springer, 492–511. https://doi.org/10.1007/978-3-642-37036-6_27

[36] Radha Jagadeesan and James Riely. 2014. Between Linearizability and Quiescent Consistency - Quantitative Quiescent Consistency. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 8573).* Springer, 220–231. https://doi.org/10.1007/978-3-662-43951-7_19

[37] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016.* ACM, 256–269. https://doi.org/10.1145/2951913.2951943

[38] 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. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[39] 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. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

[40] 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 *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.* ACM, 637–650. https://doi.org/10.1145/2676726.2676980

[41] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74).* Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

[42] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* ACM, 175–189. https://doi.org/10.1145/3009837.3009850

[43] Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew J. Parkinson. 2017. Proving Linearizability Using Partial Orders. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201).* Springer, 639–667. https://doi.org/10.1007/978-3-662-54434-1_24

[44] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201).* Springer, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[45] Siddharth Krishna, Michael Emmi, Constantin Enea, and Dejan Jovanovic. 2020. Verifying Visibility-Based Weak Consistency. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science).* Springer, 280–307. https://doi.org/10.1007/978-3-030-44914-8_11

[46] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A relational model of types-and-effects in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* ACM, 218–231. https://doi.org/10.1145/3009837.3009877

[47] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016.* ACM, 649–662. https://doi.org/10.1145/2837614.2837643

[48] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017.* ACM, 618–632. https://doi.org/10.1145/3062341.3062352

[49] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[50] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. 2013. Correct and efficient work-stealing for weak memory

models. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*. ACM, 69–80. https://doi.org/10.1145/2442516.2442524

[51] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global optimizations in relaxed memory concurrency. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. ACM, 362–376. https://doi.org/10.1145/3385412.3386010

[52] Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM, 459–470. https://doi.org/10.1145/2491956.2462189

[53] Glen Mével and Jacques-Henri Jourdan. 2021. Formal verification of a concurrent bounded queue in a weak memory model. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473571

[54] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: a concurrent separation logic for multicore OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 96:1–96:29. https://doi.org/10.1145/3408978

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

[56] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*. ACM, 267–275. https://doi.org/10.1145/248052.248106

[57] Gil Neiger. 1994. Set-Linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*. ACM, 396. https://doi.org/10.1145/197917.198176

[58] Joakim Öhman and Aleksandar Nanevski. 2022. Visibility reasoning for concurrent snapshot algorithms. *Proc. ACM Program. Lang.* 6, POPL, 1–30. https://doi.org/10.1145/3498694

[59] Anton Podkopaev, Ilya Sergey, and Aleksandar Nanevski. 2016. Operational Aspects of C/C++ Concurrency. *CoRR* abs/1606.01400 (2016). http://arxiv.org/abs/1606.01400

[60] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107

[61] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: a simpler and faster operational concurrency model. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, 1–15. https://doi.org/10.1145/3314221.3314624

[62] Azalea Raad, Marko Doko, Lovro Rozic, Ori Lahav, and Viktor Vafeiadis. 2019. On library correctness under weak memory consistency: Specifying and verifying concurrent libraries under declarative consistency models. *Proc. ACM Program. Lang.* 3, POPL (2019), 68:1–68:31. https://doi.org/10.1145/3290381

[63] William Scherer, Doug Lea, and Michael Scott. 2005. A scalable elimination-based exchange channel. (2005).

[64] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*. Springer, 333–358. https://doi.org/10.1007/978-3-662-46669-8_14

[65] Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. 2016. Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. ACM, 92–110. https://doi.org/10.1145/2983990.2983999

[66] Robert C. Steinke and Gary J. Nutt. 2004. A unified theory of shared memory consistency. *J. ACM* 51, 5 (2004), 800–849. https://doi.org/10.1145/1017460.1017464

[67] Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*. Springer, 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

[68] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A Separation Logic for a Promising Semantics. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*. Springer, 357–384. https://doi.org/10.1007/978-3-319-89884-1_13

[69] Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. 2015. Verifying read-copy-update in a logic for weak memory. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. ACM, 110–120. https://doi.org/10.1145/2737924.2737992

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

[71] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. ACM, 377–390. https://doi.org/10.1145/2500365.2500600

[72] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. ACM, 691–707. https://doi.org/10.1145/2660193.2660243

[73] Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: A program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. ACM, 867–884. https://doi.org/10.1145/2509136.2509532

[74] Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shuyu Guo. 2020. Repairing and mechanising the JavaScript relaxed memory model. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. ACM, 346–361. https://doi.org/10.1145/3385412.3385973