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:

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, 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 [33] 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) 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 LA\textsubscript{abs} 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 LA\textsubscript{abs}\textsubscript{hb} spec style (§3.1), which suffices to verify the MP example in Figure 1. We then consider the LA\textsubscript{hb} (§3.2) and LA\textsubscript{hist}\textsubscript{hb} (§3.3) styles, a weakening and a strengthening of LA\textsubscript{abs}\textsubscript{hb}, respectively. LA\textsubscript{hist}\textsubscript{hb} strengthens LA\textsubscript{abs}\textsubscript{hb} with a linearizable history

\begin{verbatim}
enq(q, 41); |
| enq(q, 42); |
| flag := rel 1 |

\end{verbatim}

\begin{verbatim}
while (\textasciitilde accflag ::= 0){
| deq(q) |
| deq(q) |
| // return 41 or 42, not empty |

\end{verbatim}

Figure 1. A Message-Passing (MP) client with Queues
to give tighter specs for stronger implementations. Meanwhile LAT$\text{ab}$ 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., po ∪ 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 0.1KLOC and 0.5 KLOC 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.

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 $\text{enq}(q, v)$ can run safely as long as it has 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 Queue$(q, vs)$ as arbitrary resources that it specifically needs. But from the perspective of clients, 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 $\text{enq}(q, v)$ 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 $\text{deq}(q)$ 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

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

²In 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, \( \text{deq}(q) \) returns empty \( e \) and the fact that the abstract state—both before and after the operation—is empty \( \{ \} \).

That an operation is allowed to consume the queue ownership for the whole duration of its execution is what makes the spec sequential: a group of threads cannot access the ownership \( \text{Queue}(q, vs) \) 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 \( (P) e (Q) \), with angle brackets \( ⟨ \) instead of curly braces. The intuitive interpretation is also a bit more subtle than normal Hoare triples: \( (P) e (Q) \) means that there exists a commit point (instruction) \( e \) 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 \( ⟨ \) . Less obvious is the quantification of \( vs \) in the precondition \( (vs. \text{Queue}(q, vs)) \): 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 \( \text{enq}(q, v) \) can withstand arbitrary concurrent updates to the state \( vs \) of \( q \) up until the commit point when it atomically transforms \( \text{Queue}(q, vs) \) (where \( vs \) is the state at that instant) to the new state \( \text{Queue}(q, vs ++ [v]) \). In contrast, the sequential spec \( \text{SEQ-Enq} \) implicitly quantifies over \( vs \) with a normal universal quantifier \( \forall vs \) at the outside: this allows the implementation to assume exclusive ownership of \( \text{Queue}(q, vs) \) for an arbitrary but unchanging \( vs \), thereby prohibiting concurrent interference.

Last but not least, we add a local precondition \( \text{isQueue}(q) \), 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 \( \text{Queue}(q, vs) \) which is neither duplicable nor local.

Intuitively, it should be clear that \( (P) e (Q) \) 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 \( \text{Queue}(q, vs) \)? 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 \( I \) asserts the existence of \( I \) in the global invariant space. (The “later” modality * is an artifact of the step-indexed model of Iris, which we will gloss over.)

\[
\text{Inv-Acc} \quad \langle I * P \rangle \{ c \{ I * Q \} \}
\]

\[
\text{LAInv-Acc} \quad \langle I * P \rangle \{ e \{ I * Q \} \}
\]

The LAT invariant access rule \( \text{LAInv-Acc} \) strengthens \( \text{Inv-Acc} \), 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., \( \exists vs_1, vs_2. \text{Queue}(q_1, vs_1) * \text{Queue}(q_2, vs_2) * R(vs_1, vs_2) \), 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: View := Loc \( \rightarrow \) Time. The timestamps are indices in 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**

\[
\{ \exists V \cdot \ell \mapsto h \} \quad \ell := \text{rel} v \quad \left( \begin{array}{l}
\exists t \notin h, V', V(t) < t \\
V \cup \{ \ell \mapsto t \} \subseteq V' \\
\exists V' \cdot \ell \mapsto h[t \mapsto (v, V')]
\end{array} \right)
\]

**Acq-Read**

\[
\{ \exists V \cdot \ell \mapsto h \} \quad \text{acq} \ell \quad v \cdot \exists t. V(t) \leq t \land h(t) = (v, V') \quad \left( \begin{array}{l}
\exists (V \cup V') \cdot \ell \mapsto h
\end{array} \right)
\]

Both rules concern (1) a persistent fact \( \exists V \) (read “seen \( V' \)”) that the executing thread \( \pi \)'s current local view is at least \( V \), and (2) the atomic points-to ownership \( \ell \mapsto h \) of the location \( \ell \) that is the thread writing to/reading from. The atomic points-to includes a *history* \( h \) (in Time \( \mapsto % Val \times View) of \( \ell \) 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 \( \pi \)'s view after the instruction, as encoded in \( \exists V' \), 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 \( V_1 \subseteq V_2 := \forall (V_1(\ell) \leq V_2(\ell)) \). 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 \( \pi \) releases its observations (its local view \( V \)) through the write. Another thread, say \( \rho \), 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 \( \exists (V \cup V') \). As such, the release-acquire synchronization between \( \pi \)'s release write and \( \rho \)'s acquire read is reflected in the logic by \( \pi \)'s sending its view \( V \) to \( \rho \). Intuitively, any operation that happens-before \( \pi \)'s release write is observed in \( V' \), and therefore also observed by \( \rho \)'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 \( \exists V \) 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
Figure 2. Specifications of Queue operations, from stronger to weaker memory consistency models.

is why we call them LAT\textsuperscript{abs}_0 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 LAT\textsuperscript{abs}_0 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 $\epsilon$ 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 threadify differently, so it may be that the thread $\pi$ sees the queue as empty and returns $\epsilon$, but the queue is in fact not empty, because a fresh enqueue by another thread $\rho$ has not become visible to $\pi$ 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 LAT\textsuperscript{abs}_0 style which generalizes the LAT\textsuperscript{abs}_0 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 \( \text{LAT}_{\text{hb}}^{\text{abs}} \) and \( \text{LAT}_{\text{hb}}^{\text{hist}} \) spec styles, a weakening and a strengthening of \( \text{LAT}_{\text{hb}}^{\text{abs}} \), respectively.

### 3.1 Graph-Based Specs to Encode Partial Orders

The \( \text{LAT}_{\text{hb}}^{\text{abs}} \) style extends the \( \text{LAT}_{\text{so}}^{\text{abs}} \) style by exposing a greater part of \( \text{hb} \). An instance for queues is given in \( \text{Abs-Hb-Enq} \), \( \text{Abs-Hb-Deq} \), and \( \text{Abs-Hb-Queue-Consistency} \) (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 \( \text{Abs-Hb-Deq} \)’s failure case, where the caller sees the queue as empty. Here, the spec provides more information about how the resulting read-only empty \( \text{dequeue} \) operation is ordered with other operations in \( \text{hb} \).

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 \( G \in \text{Graph} \) 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 \( e \in \text{EventId} \) to event data of type \( \text{Event} \), and (2) a set of event id pairs that encodes the \( \text{so} \) relation. We use \( G(e) \) to denote the event data for \( e \) in \( G \), and \( G.so \) to denote the \( \text{so} \) relation of \( G \).

The type \( \text{Event} \) is a tuple of (1) an event type (\( \text{type} \)), (2) a physical view (\( \text{view} \)), and (3) a logical view (\( \text{logview} \)). In Figure 2 we give an instance of the event type for queues: the events can be an \( \text{enqueue} \) event of \( v \) (\( \text{Enq}(v) \)), a successful \( \text{dequeue} \) event of \( v \) (\( \text{Deq}(v) \)), or a failing (empty) \( \text{dequeue} \) event (\( \text{Deq}(e) \)). 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., \( e \in G(d).\text{logview} \), 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 \( \text{hb} \) relation between library operations, just as the physical view construction is an approximation of \( \text{hb} \) between memory instructions. The difference is that while physical views approximate \( \text{hb} \) globally between library instructions, logical views only approximate \( \text{hb} \) locally for the library in question. As such, our logical views correspond to the \text{local happens-before} \( \text{hb} \) relation of a library object introduced by Yacovet. Henceforth we use \( e \in G(d).\text{logview} \) and \( (e, d) \in G.lhb \) interchangeably.

The \( \text{LAT}_{\text{hb}}^{\text{abs}} \) style extends \( \text{LAT}_{\text{so}}^{\text{abs}} \) 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 \( \text{Abs-Hb-Enq} \), when an enqueue of \( v \) commits, the current graph \( G \) of \( q \) is extended atomically with a fresh event \( e \) whose type is \( \text{Enq}(v) \), 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 \( \text{Abs-Hb-Enq} \), \( G.lhb \) extends \( G.lhb \) by setting \( G'(e).\text{logview} = M' \), the set containing all operations that happen before \( e \). \( M' \) includes \( M_0 \)—the \text{local logical view} of the calling thread, which tracks the operations that happen-before the \( \text{enq} \) call. This tracking of thread-local logical views is done by a new persistent assertion \( \text{SeenQueue}(q, G_0, M_0) \), 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 \( \text{SeenQueue}(q, G', M') \) 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 \( \text{SeenQueue} \) 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 \( \text{Abs-Hb-Deq} \) to verify the MP client in Figure 1.

Compared to the \( \text{LAT}_{\text{so}}^{\text{abs}} \) style, in \( \text{LAT}_{\text{so}}^{\text{abs}} \) each library type has a local logical view assertion like \( \text{SeenQueue} \) 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 \( \text{isQueue}(q) \) assertion in \( \text{Abs-So-Enq} \). 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 \( \text{LAT}_{\text{hb}}^{\text{abs}} \) style specifies properties of the abstract state and the partial orders through the library’s \text{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 \textit{Abs-Hb-Queue-Consistency}: the queue ownership assertion \textit{Queue}(q, vs, G), which is consumed and reproduced around the commit point, always implies consistency. So when \textit{Abs-Hb-Enq} and \textit{Abs-Hb-Deq} extend \textit{(vs, G)} to new state \textit{(vs′, G′)}, the operations can assume QueueConsistent\textit{(vs, G)} and must then re-establish QueueConsistent\textit{(vs′, G′)}.

More specifically, if \textit{deq} succeeds with a value \textit{v}, \textit{Abs-Hb-Deq} tells the client that \textit{G.so} extends \textit{G.so} with a new pair \textit{(e, d)} where \textit{d} is the new successful event added by the dequeue operation and \textit{e} is an existing queue event that \textit{d} dequeues from. Therefore, through \textit{Abs-Hb-Queue-Consistency}, the spec additionally says that \textit{(e, d)} satisfies, among other things,

1. \textit{Queue-Matches}: the return value \textit{v} of the dequeue \textit{d} must match the value enqueued by \textit{e}.
2. \textit{Queue-FIFO}: if there is another enqueue event \textit{e′} that happens before \textit{e}, then \textit{e′} must already have been dequeued by some \textit{d′ (\textit{(e′, d′)} ∈ G.so)}, and our \textit{d} cannot happen before \textit{d′ (\textit{(d, d′)} ∈ G.lhb)}. (The consistency conditions on enqueue events are elided, so we will not discuss them.)

\textbf{Weaker but flexible.} The Queue-FIFO condition appears weaker than what one might expect, i.e., \textit{(d′, d)} ∈ G.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 deques. In fact, we have verified that Queue-FIFO is satisfiable by a fairly relaxed implementation (similar to the weak version in [62]) of the Herhy-Wing queue [34]. The implementation ensures lhb only between matching enqueue-dequeue pairs, but not among enqueues or among deques. (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., \textit{(d′, d)} ∈ G.lhb ∨ (d, d′) ∈ G.lhb, and can thus exclude the right-hand side of the disjunction and regain the stronger FIFO condition with \textit{(d′, d)} ∈ G.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.

\textbf{Message-Passing client verification.} When a call to \textit{deq} returns empty (\textit{e}), consistency demands that the added empty dequeue event \textit{d} satisfies \textit{Queue-EmpDeq}, which is sufficient to verify the MP client (Figure 1). Intuitively, \textit{Queue-EmpDeq} says that there cannot be another enqueue \textit{e} which happens before \textit{d} but has not been dequeued in \textit{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 \textit{e1} and \textit{e2} 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 \textit{Queue-EmpDeq} the dequeue cannot be an empty one and must dequeue from \textit{e1} or \textit{e2} and return either 41 or 42.

The proof sketch of this example in Compass is given in Figure 3. The following pattern mentioned at the end of §2.2, we put the ownership \textit{Queue(q, _)} in an invariant to enforce a concurrent protocol on the queue, using a dequeue permission called \textit{deqPerm} that can be defined with Iris ghost state [38]. One dequeue permission \textit{deqPerm(1)} is needed to perform one successful dequeue. This requirement can be seen in the invariant: \textit{deqPerm(size(G.so))} counts the number of successful deques, and a successful dequeue will extend \textit{G.so} by 1, so anyone who successfully deques needs to put in a \textit{deqPerm(1)} to re-establish the invariant. For our particular example, we also implement \textit{deqPerm} such that there are only two \textit{deqPerm(1)’s} (i.e., \textit{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 \textit{SeenQueue(q, ∅, θ)} of the initial empty state.

The verification of the left-most thread is straightforward: for each enqueue, we use \textit{LAINV-Acc} to open the invariant and then use \textit{Abs-Hb-Enq}. Afterwards the thread has two enqueue events \textit{(e1, e2)} in its logical view, and the write to flag releases \textit{SeenQueue(q, G1, \{e1, e2\})} to the right-most thread. The verification of the middle thread uses \textit{LAINV-Acc} and \textit{Abs-Hb-Deq}, and if the dequeue succeeds, \textit{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 \textit{SeenQueue(q, G1, \{e1, e2\})} from the left-most thread. We then use \textit{LAINV-Acc} and \textit{Abs-Hb-Deq} to perform the dequeue, with \textit{M0 := \{e1, e2\}}. Before re-establishing the invariant, we inspect the resulting dequeue \textit{d1}. If it is a successful dequeue, we can put \textit{deqPerm(1)} in the invariant and finish. If \textit{d1} is an empty dequeue, we derive a contradiction. As there are only two \textit{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 \textit{d1}) graph \textit{G} can have at most one dequeue: \textit{size(G.so)} ≤ 1. Furthermore, the thread has observed two enqueues, so in \textit{G} there must be at least one.

\footnote{For example, an element can only be dequeued once.}
enqueue that is not dequeued yet, which must be in \(\{e_1, e_2\}\). Due to \(\text{SeenQueue}(q, G, \{e_1, e_2\})\), both \(e_1\) and \(e_2\) happen before \(d_1\). By \(\text{Queue-EmpDeq}\), we have our contradiction.

### 3.2 Weaker Specs by Abandoning Abstract States

The \(\text{LAT}^\text{abs}_{\text{hb}}\) 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 \(\text{LAT}^\text{abs}_{\text{hb}}\) specs for queues (and therefore transitively the \(\text{LAT}^\text{abs}_{\text{sc}}\) 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 \(v\) 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 reasoning 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 \(\text{LAT}^\text{lb}_{\text{hb}}\) specs, a weakening of the \(\text{LAT}^\text{abs}_{\text{sc}}\) specs where the abstract state is abandoned. In particular, our instance of the \(\text{LAT}^\text{lb}_{\text{hb}}\) specs for queues is exactly the specs \(\text{Abs-Hb-Enq}\) and \(\text{Abs-Hb-Deq}\) (Figure 2) without \(v\).

\(\text{LAT}^\text{lb}_{\text{hb}}\) 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:

\[
\begin{align*}
\text{SeenQueue}(q, \_ \_ \_ ) & \ast \quad \text{consuming} & \text{SeenQueue}(q, \_ \_ \_ ) & \ast \\
\{a_p \mapsto [a_0, \ldots, a_{n-1}] \ast \ldots\} & \quad & \{a_c \mapsto [0, \ldots, 0] \ast \ldots\} & \quad & \text{consume} & \{q, a_c, 0, n\} & \ast \\
\{a_p \mapsto [a_0, \ldots, a_{n-1}] \ast \ldots\} & \quad & \{a_c \mapsto [a_0, \ldots, a_{n-1}] \ast \ldots\} & \quad & \text{produce} & \{q, a_p, 0, n\} & \ast \\
\end{align*}
\]

Here, there is only one thread performing enqueues—the producer—and only one thread performing dequeues—the consumer. The producer reads the array \(a_p\) for elements with the indices in \([0, n]\) and enqueues them in that order, while the consumer keeps dequeuing for \(n\) elements and writes them in the indices \([0, n]\) of the array \(a_c\) in the dequeuing 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 \(\text{LAT}^\text{lb}_{\text{hb}}\) specs for queues (i.e., \(\text{Abs-Hb-Enq}\) and \(\text{Abs-Hb-Deq}\) without abstract states) to derive the stronger \(\text{LAT}^\text{lb}_{\text{hb}}\)-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 \(\text{LAT}^\text{lb}_{\text{hb}}\) 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 \(\text{LAT}^\text{hist}_{\text{hb}}\) style, and it is a strengthening of the \(\text{LAT}^\text{abs}_{\text{sc}}\) specs with a linearizable history \(H\). An excerpt of the instance for \(\text{stacks}\) is given in Figure 4. The linearizable history \(H\) subsumes both the event graph \(G\) and the abstract state \(v\): 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 \(\text{lb}\), but only needs to respect \(\text{lb}\) (i.e., \(H.\text{lb} \subseteq to\)). Additionally, \(to\) directly encodes the stack’s LIFO property, as well as stricter behaviors of failing empty pop operations, as required by \(\text{Hist-Hb-Stack-Lineartizable}\). That is, \(to\) is a reordering (permutation) of \(H\)’s operations, satisfying \(\text{interp}(to, vs)\) 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

\[\text{Note that the concept for histories of library operations is different from that for histories of write events to locations in §2.3.}\]
We can then construct a verification challenge is the construction of specs to verify an RMC implementation of the elimination stack. This verification is both a linearizability-style LAT 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, v₁) on the exchange object x with some value v₁ ≠ ⊥. If the return value is ⊥, the exchange has failed, but if it is some v₂ ≠ ⊥, then the thread has successfully exchanged v₁ for v₂ with another thread. Additionally, the two threads synchronize with each other, 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:

```latex
\begin{align*}
\text{try\_push}(s, v) &::= \text{if try\_push}'(s, \text{base}, v) \text{ then true}
\text{else exchange}(s, \text{ex}, v) \text{ == SENTINEL}
\text{try\_pop}(s) &::= \text{let } v = \text{try\_pop}'(s, \text{base}) \text{ in}
\text{if } v \neq \text{FAIL\_RACE} \text{ then v'}
\text{else } \text{exchange}(s, \text{ex}, \text{SENTINEL}) \text{ in}
\text{if } v' \notin \{\text{SENTINEL}, \bot\} \text{ then v'}
\text{else FAILURE}
\end{align*}
```

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 e if try\_pop’ does.) Otherwise, try\_pop calls exchange with SENTINEL, and only succeeds with the returned value v’ if it is matched (v’ ≠ ⊥) with a push (v’ ≠ SENTINEL).

**Verification results.** Assuming the LAT\_hb\_spec specs for the base stack and the exchanger (Figure 5), we have verified that our relaxed ES implementation satisfies the same LAT\_hb.
whenever a base operation commits, the ES operation needs A simplified Treiber stack and exchanger implementations satisfy their exceed together with another exchange identified by Exchange change fails, the return value v Exchange rent graph x an atomically shared ownership assertion cal logical view assertion SeenExchanges 4.2 Strong Specs for Exchangers the exchanger specs next. 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 Lat_{bb} style spec for the exchange function is shown in HB-EXCHANGE (Figure 5). The spec involves a local logical view assertion SeenExchanges(x, G, M) and an atomically shared ownership assertion Exchanger(x, G) for the exchanger object x. At the commit point, the current graph G is extended with a new event e1 with type Exchange(v1, v2), where v2 is the returned value. If the exchange fails, the return value v2 is ⊥ and the event type is Exchange(v1, ⊥). If the exchange succeeds, it can only succeed together with another exchange identified by e2 and the G.so relation is extended with the two events in both directions {((e1, e2), (e2, e1))}, 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 ofLATs 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, {...}) 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 event e1 is committed before the other event e2, i.e., e1 < e2, then e2 is the helper. Otherwise, if e2 < e1, then e1 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 e1 is the helpee, it only adds itself to the current graph G: G′ = G[e1 ⇝ Exchange(v1, v2), V1, M′]; but if e2 is the helper, it knows that the helpee’s event e1 must already be in the current graph: G(e2) = Exchange(v2, v1), V2, M′), and the helper not only adds itself to the current graph G, but also extends G.so with the pairs {((e1, e2), (e2, e1))}. The client thread of the helper learns all of this information about the updated G′ atomically right after the helpee’s commit, by which point it has also locally observed both e1 and e2, via SeenExchanges(x, G′, M′) and {e1, e2} ⊆ M′. The client thread of the helper, on the other hand, right after its own commit has only locally observed its own event e1, via SeenExchanges(x, G′, M′ \ {e1}), because the helper commit has not been performed yet and e2 has not been added to G′. Only in the local postcondition (in {...}), after both
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 \(\text{SeenExchanges}(x, G'', M')\).

**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 \(\text{Exchanger}(x, G)\) does not imply \(\text{ExchangerConsistent}(G)\). Instead, we have \(\text{ExchangerConsistent}(G')\) 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 \(\text{LAInv-Acc}\) 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.

\[
\text{OINV-ALLOC} \quad \text{objective}(I) \quad \Rightarrow \quad I
\]

That is, if we have \(\triangleright I\), we can put it in an invariant as long as we can show that it is objective, i.e., objective(\(I\)). 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 \(\exists V\) says that its owner's observations are at least \(V\). Another important modality is the view-at modality \(\forall_V P\) 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: \(\text{objective}(\forall_V P) \quad \Rightarrow \quad \exists V, \forall V \ast \forall_V P \quad \Rightarrow \quad \forall V \ast \forall_V P \quad \Rightarrow \quad \forall_V P\)
- VA-ELIM: \(\forall_V P\) 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 (\(\exists V\)). Having done this, we can send \(\forall_V P\) and \(\exists V\) away on different routes: \(\forall_V P\) can be put inside an invariant, and \(\exists V\) 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 (\(\exists V\)), 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.

\[
\text{AT-REL-WRITE} \quad \{ \exists V \ast \ell \exists h_0 \ast \forall_v \ell \rightarrow_{at} h \} \quad \ell := \text{rel } v
\]

\[
\{ \ell, \exists t \notin h \exists h_0 \ast \forall_v \ell \rightarrow_{at} h \ast \exists t \ast \{ V \cup \{ \ell \rightarrow t \} \subseteq V \ast \exists V' \ast \exists t \ast \ell \exists h_0 [t \mapsto (v, V')] \ast \forall_v \ell \rightarrow_{at} h[t \mapsto (v, V')] \}
\]

\[
\text{AT-ACQ-READ} \quad \{ \exists V \ast \ell \exists h_0 \ast \forall_v \ell \rightarrow_{at} h \} \quad \ast \text{acq } \ell
\]

\[
\{ \forall V \ast \ell \exists h_0 \ast \forall_v \ell \rightarrow_{at} h \ast \text{acq } \ell \}
\]

\[
\{ \forall V \ast \ell \exists h_0 \ast \forall_v \ell \rightarrow_{at} h \ast \text{acq } \ell \}
\]

\[
\{ \forall V \ast \ell \exists h_0 \ast \forall_v \ell \rightarrow_{at} h \ast \text{acq } \ell \}
\]

In these rules, we do not require the ownership of atomic points-to \(\ell \rightarrow_{at} h\) locally. Instead, we need it only objectively, under a view-at modality at some view \(V_h\), but together with some local history observation \(\ell \exists h_0 \ast \text{at } h\) 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 \(V_h \sqcup V'\) 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 LAT 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 $V_q$. This makes $\text{Queue}(q, G)$ an objective resource that can be shared concurrently, and yet $\text{Queue}(q, G)$ 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 $\text{LAT}^{\text{new}}$ 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 non-trivial 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 “Rust-Belt” 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).
Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Radha Jagadeesan and James Riely. 2014. Between Linearizability and
Radha Jagadeesan, Gustavo Petri, Corin Pitcher, and James Riely. 2013. Modular
Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. 2015. Modular
Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M.
Michael Emmi and Constantin Enea. 2019. Weak-consistency speci-


