Mechanized Verification of a Fine-Grained Concurrent Queue from Facebook’s Folly Library

SIMON FRIIS VINDUM, Aarhus University, Denmark
DAN FRUMIN, University of Groningen, Netherlands
LARS BIRKEDAL, Aarhus University, Denmark

We present the first formal specification and verification of the fine-grained concurrent Multi-Producer-Multi-Consumer Queue algorithm from Facebook’s C++ library Folly of core infrastructure components. The queue is highly optimized, practical, and used by Facebook in production where it scales to thousands of consumer and producer threads. We present an implementation of the algorithm in an ML-like language and formally prove that it is a contextual refinement of a simple coarse-grained queue (a property which implies that the MPMC queue is linearizable). We use the ReLoC relational logic and the Iris program logic to carry out the proof and to mechanize it in the Coq proof assistant. The MPMC queue is implemented using three modules, and our proof is similarly modular. By using ReLoC and Iris’s support for modular reasoning we verify each module in isolation and compose these together. A key challenge of the MPMC queue is that it has a so-called external linearization point, which ReLoC has no support for reasoning about. Thus we extend ReLoC, both on paper and in Coq, with novel support for reasoning about external linearization points.

ACM Reference Format:

1 INTRODUCTION

It is well-known that it is challenging to program, specify, and verify fine-grained concurrent algorithms, and in recent years we have seen much progress on program logics for specifying and verifying such algorithms, e.g., [7, 13, 20, 24, 28, 33–39].

In this paper, we present the first formal specification and verification of the highly efficient and practical concurrent Multi-Producer-Multi-Consumer Queue algorithm found in Facebook’s open source library Folly (or, simply, the MPMC queue in the rest of the paper). The Folly library is an open-source collection of key infrastructure components implemented in C++ and used extensively in production at Facebook [11]. The library contains, among many other things, the MPMC queue. The queue was originally developed by Nathan Bronson for the purpose of connecting two thread pools inside TAO [4], Facebook’s distributed data store for their social graph. One of the key ideas used in the algorithm is to improve scalability by decreasing the contention found in other lock-free algorithms, such as the Michael-Scott queue [30], by striping the queue across q “smaller” sub-queues. To avoid the overhead of maintaining q sub-queues, the striping is taken to the extreme

1The source code is available online at https://github.com/facebook/folly/blob/master/folly/MPMCQueue.h.

Authors’ addresses: Simon Friis Vindum, vindum@cs.au.dk, Aarhus University, Denmark; Dan Frumin, d.frumin@rug.nl, University of Groningen, Netherlands; Lars Birkedal, birkedal@cs.au.dk, Aarhus University, Denmark.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.
XXXX-XXXX/2021/3-ART $15.00
https://doi.org/10.1145/nnnnnn.nnnnnnn

, Vol. 1, No. 1, Article . Publication date: March 2021.
by letting each sub-queue store only a single element. These single-element queues, can then be simpler and faster. In fact they are implemented merely as a reference to a value and a turn sequencer. The latter is just a reference to an integer and serves as a synchronization mechanism used by the single-element queue to guard access to its value. The enqueue and dequeue operations on the MPMC queue are delegated to one of the single-element queues by taking a ticket from one of two ticket dispenser using an atomic increment (FAA). After receiving a ticket, up to q separate enqueue or dequeue operations can proceed in parallel, completely independent of each other as they operate on different single-element queues. The FAA instruction thus become the main point of contention, but since an FAA instruction (unlike CAS) always succeeds, this design, in the words of Bronson, “makes contention count” as its cost always pays off in significant progress being made in the algorithm [3]. Altogether, this makes the queue scalable to hundreds of thousands of producer and consumer threads.

We present an implementation of the MPMC queue, the single-element queue, and the turn sequencer in a strongly-typed ML-like language with concurrency primitives. The implementation captures the essence and the key verification challenges of the algorithm while eliding some of the low-level details of the original C++ implementation. We prove that the MPMC queue contextually refines a coarse-grained concurrent queue. The coarse-grained queue uses a lock and allows only one thread at a time to access the queue in any way. Informally, the contextual refinement property then means that in any program we may replace uses of the simple “obviously correct” coarse-grained concurrent queue with the more efficient, but also more complicated, MPMC queue, without changing the observable behaviour of the program. More precisely, an expression e contextually refines another expression e’, denoted Δ; Γ ⊢ e ≲Ctx e’ : τ, if for all contexts C of ground type, if C[e] terminates with a value, then there exists an execution of C[e’] that terminates with the same value. To show the contextual refinement, we use the recently proposed relational logic ReLoC [13, 14]. ReLoC is implemented on top of the Iris program logic [21, 22, 24, 26] and greatly simplifies proofs of contextual refinement by offering rules that allows one to reason about refinements at a high level of abstraction. This results in proofs that are simpler and shorter than other approaches (e.g., directly using step-indexed Kripke logical relations as in [36]). Another reason why we have chosen to use ReLoC and Iris for our verification is that both logics are fully formalized in the Coq proof assistant and come equipped with a so-called proof mode [25, 27] that allows one to reason formally and interactively in ReLoC and Iris using tactics, much as one does in Coq. This allowed us to interactively develop a machine-checked proof.

Another popular correctness criterion for concurrent algorithms is linearizability [18], briefly, the property that all operations appear to take place atomically at some point during their execution. Linearizability, however, has mainly been considered for first-order languages and with certain restrictions placed on how clients can interact with the concurrent algorithm. In such a setting contextual refinement and linearizability are equivalent [12], and linearizability can thus be used as a proof method to show contextual refinement. However, by using ReLoC to prove contextual refinement directly, we get a stronger result that applies in a more realistic language and setting. The client can fork any number of threads (since the language has fork-based concurrency), it can store arbitrary values on the queue (since the language has universally quantified types and higher-order store), and interact directly with the the queue as a first-class value to construct arbitrarily many instances of the queue. In this more permissive setting, linearizability is not equivalent to contextual refinement, as the latter only holds if the queue properly encapsulates its implementation details. In our case, we achieve encapsulation via the common approach of using a closure (relying on the language having higher-order functions) to hide the queues internal state in a closure. Indeed, to the best of our knowledge, linearizability has not been considered for a higher-order language...
Mechanized Verification of Folly Queue

with all the features our language has (fork-based concurrency, higher-order store, universal and existential types).

To verify a fine-grained concurrent algorithm, one of the key steps is to identify the *linearization points* of its operations. Intuitively, a linearization point for an operation is the point during execution where the operation “appears to take place”. In our analysis of the MPMC queue we discover that, in some cases, the linearization point of dequeue is *external*. A linearization point is external if it happens during the execution of another operation. For dequeue its linearization point may happen within the execution of enqueue. This is not immediately obvious by looking at the code. External linearization points typically occur in the presence of *helping* (code in one operation that carries out work for a different operation), but the MPMC queue contains no such helping. Instead, as we explain in detail later, the external linearization points arises because the algorithm, in contrast to other fine-grained concurrent queues, is not entirely non-blocking: if the all the single-element queues are full (resp. empty) then enqueue (resp. dequeue) is blocked.

One may categorize linearization points into three classes [9]: fixed, future-dependent, and external. The first version of ReLoC [13] had support for reasoning about fixed linearization points only, and ReLoC Reloaded [14] added support for future-dependent linearization points, through its use of Iris-style prophecy variables [23]. However, we observe that neither version of ReLoC supports reasoning about external linearization points. Hence, to verify the MPMC queue we extend ReLoc with new proof rules and generalize existing proof rules to be able to reason about external linearization points. Our extension thus “completes the picture” by making ReLoC able to handle all three classes of linearization points.

As mention, the MPMC queue is implemented as three submodules. In particular, the MPMC queue is implemented using the single-element queue, which is implemented using the turn-sequencer. A strength of our approach that our contextual refinement proof is similarly modular: it makes use of (unary) Hoare-style specifications of the turn-sequencer and the single-element queue. We here leverage that ReLoC allows for compositional modular reasoning and that it, following [35], includes a proof rule that allows one to use Hoare-style specifications, written in the Iris program logic, to simplify reasoning about the left-hand-side in a relational proof [13, Section 7.4]. We thus end up not only with a refinement proof of the MPMC queue, but also with reusable specifications for the single-element queue and the turn sequencer.

In order to arrive at sufficiently composable specifications we make use of a, to the best of our knowledge, novel proof pattern involving a resource algebra over *infinite* sets, to keep track of which turns are “still available” (see Section 4). The idea of using infinite structures to improve composability is well-known in the context of functional programming [19]. In our case, the specification of the turn-sequencer supplies its client with an *infinite* set of turns and the specification of the single-element queue gives its client *two* infinite sets of tickets. The proof of the latter relies on being able to split the underlying turn sequencer’s infinite set of turns into *two* infinite sets of turn. This approach greatly simplifies the proofs and makes it possible to reason about the single-element queue in the refinement proof with the details of the turn sequencer having been abstracted away. We believe this proof pattern could also be used to simplify reasoning about other algorithms based on these components, and have used it to additionally verify a ticket lock based on the turn sequencer.

Another challenge in verifying the MPMC queue is that its physical state (i.e., the actual content in the underlying array) does not immediately determine the *abstract state* of the queue (i.e., the state that is observable through the queue interface). In particular, a value may be present in the physical state of the queue without actually being in the queue (i.e., not observable with a dequeue operation). The reverse is also the case, a value can be part of the abstract state of the queue, without being present anywhere in the physical state. This means that the common approach of relating a
concurrent algorithm’s physical state to its abstract state by using a representation predicate does not apply. Instead, we relate the abstract state of the queue to \textit{ghost state} which is only tied to the physical state of the two ticket dispensers.

In summary, we believe that verifying the MPMC queue serves as an interesting example, as it is challenging to verify, used at scale in the industry, and has not been treated in the literature before. It is also good example of a concurrent algorithm where modular reasoning comes to its right: the MPMC queue has a layered implementation, composed of several modules. Our proof mirrors this composition of submodules, and builds on it by providing sufficiently modular specifications for each of the submodules. The MPMC queue is also an informative case-study that provides motivation for extending ReLoC with support for reasoning about external linearization points.

\textbf{Outline and contributions.} We make the following contributions in the paper:

- Since the MPMC queue has not been described in the literature before, we give a detailed explanation of it (Section 2).
- We informally analyze the MPMC algorithm and identify the linearization points, and observe that one of them is an external linearization point (Section 3).
- We define and prove Hoare-style specifications for the submodules used in the implementation of the MPMC queue (Section 4).
- In order to derive those specifications, we introduce a novel proof pattern involving an encoding of infinite sets as a resource algebra used for specifying and reasoning about infinite sets of turns and tickets (Section 4.3).
- We show that the MPMC queue contextually refines a coarse-grained queue. (Sections 5, 6 and 8). Our proof is \textit{modular} and makes use of the aforementioned Hoare-style specifications for the submodules.
- We extend ReLoC, both on paper and in Coq, with support (including tactics) for reasoning about external linearization points (Section 7).
- All the results in this paper and two additional examples of algorithms with external linearization points are formalized in the Coq proof assistant (Section 9). The Coq development can be found in the accompanying files.

We discuss related and future work in Section 10.

\section{The Folly MPMC Queue}

The MPMC queue is implemented using a \textit{single-element queue}, which in turn is implemented using a \textit{turn sequencer}. We describe the three data-structures, starting with the turn sequencer and proceed bottom up.

\subsection{Turn Sequencer}

A turn sequencer is a data structure that implements mutual exclusion by \textit{sequentializing} access to a critical section among threads \textit{ordered} by a monotonically increasing turn. The turn sequencer implementation is shown in Figure 1a.

The turn sequencer provides two operations: \texttt{wait} and \texttt{complete}. These are similar to the acquire and release operations on a lock, but they take an additional natural number as an argument. The natural number specifies which \texttt{turn} to wait for or to complete. The turn sequencer guarantees that if a thread waits for the \texttt{n}th turn, then it will only proceed once all the preceding turns have been completed. In order for this to hold, the turn sequencer assumes that its clients never wait for the same turn several times. As such it is the responsibility of clients to manage the turns, \textit{i.e.}, which natural numbers they wait for. Compared to a lock, this places a greater demand on the
newTS : 1 → ref int
newTS () = ref(0)

complete : ref int → 1
complete ts turn = ts ← turn + 1; ()
wait : ref int → 1
wait ts turn =
  let turn’ = !ts in
  if (turn’ = turn) then ()
  else wait ts turn

SEQ α = ref int × ref (option α)
queueSEQ : ∀α. 1 → SEQ α
queueSEQ () = λ. (newTS (), ref(None))

enqueueSEQ : SEQ α → int → α → 1
enqueueSEQ (ts, r) enqTurn v =
  let turn = enqTurn * 2 in
  wait ts turn;
  r ← Some(v);
  complete ts turn

dequeueSEQ : SEQ α → int → α
dequeueSEQ (ts, r) deqTurn =
  let turn = deqTurn * 2 + 1 in
  wait ts turn;
  let v = match !r with
  | Some(x) → x
  | None → assert(false)
  in complete ts turn; v

queueMPMC : ∀α. int → (1 → α) × (α → 1)
queueMPMC q = λ.
  let slots = arrayInit q (queueSEQ ⟨⟩) in
  let pushTicket = ref(0) in
  let popTicket = ref(0) in
  λv. dequeue slots q pushTicket v,
  λx. dequeue slots q popTicket

enqueue : array (SEQ α) → int → int → α → 1
enqueue slots q pushTicket v =
  let t = FAA(pushTicket, 1) in
  let idx = t mod q in
  let ticket = t/q in
  enqueueSEQ slots[idx] ticket v

dequeue : array (SEQ α) → int → int → α
dequeue slots q popTicket =
  let t = FAA(popTicket, 1) in
  let idx = t mod q in
  let ticket = t/q in
  dequeueSEQ slots[idx] ticket v

(a) Turn sequencer and single-element queue.

(b) MPMC queue.

Fig. 1. Implementation of the MPCM queue.

We implement the turn sequencer as a pointer ts to a number, which represents the current turn. The function wait ts n simply spins on that pointer until its value is equal to n. The implementation of complete ends the current turn by incrementing ts.

2.2 Single-Element Queue

A single-element queue is a queue with capacity 1. Our implementation is shown in Figure 1a. It is a blocking queue: if it is empty (full) then any subsequent dequeue (enqueue) is blocked until the queue becomes non-empty (non-full). This implies that both dequeue and enqueue always succeed.

Similarly to the turn sequencer, the single-element queue’s operations take a turn as an argument. The turn argument specifies the order of the operations: an enqueue or dequeue operation is carried out only after all operations with a lower number has been carried out. For an enqueue and a dequeue operations with the same turns, the enqueue is carried out first.
queue\textsubscript{CG} : \forall \alpha. (1 \to \alpha) \times (\alpha \to 1)

\text{enqueue\textsubscript{CG}} = \Lambda.

\begin{verbatim}
let w = (newlock(), ref([])) in
(\lambda v. \text{enqueue\textsubscript{CG}} w v,
 \lambda x. \text{dequeue\textsubscript{CG}} w)
\end{verbatim}

dequeue\textsubscript{CG} : lock \times list \to \alpha

dequeue\textsubscript{CG} (lk, hd) =
acquire lk;
match ! hd with
| [] \to assert(false)
| h :: t \to hd \leftarrow t;
release lk;
\begin{verbatim}
\h
\end{verbatim}

\begin{fig}
\text{Coarse-grained queue.}
\end{fig}

The single-element queue is implemented as a reference to an option type, protected by a single turn sequencer. In order to ensure that the turn sequencer operations are called with correct turns, the implementations of the enqueue and dequeue operations adhere to the following discipline.

The even turns of the turn sequencer correspond to the enqueue operations and the odd turns correspond to the dequeue operations. Hence when enqueue\textsubscript{SEQ} (dequeue\textsubscript{SEQ}, respectively) is called with turn \( n \), the corresponding turn for the turn sequencer is \( 2n \) (\( 2n+1 \), respectively). Not only does this allow for a single turn sequencer to provide turns for both of the operations, it also ensures that the enqueue and dequeue operations are carried out in the correct order. The first enqueue gets the first even turn, 0, the first dequeue gets the first odd turn, 1, and so on. Hence the enqueue and dequeue operations alternately get access to the pointer, and the dequeue operation can be sure that a value is present when it reads the pointer.

### 2.3 MPMC queue

The MPMC queue is a blocking queue of a fixed capacity \( q \). The implementation of the MPMC queue is shown in Figure 1b. The binary operator “\( \mod \)” denotes modulo (or remainder) and “\( \div \)” denotes integer division (i.e., \( 3/2 = 1 \)).

Upon initialization, an array of length \( q \) is created, with each entry containing a single-element queue. The function arrayInit constructs an array of the given length, calls the given function once for each entry and sets the entry to the result. In addition to the array, the queue contains two ticket dispensers (references to natural numbers): pushTicket and popTicket. The first keeps track of the enqueue operation, and the second does the same for the dequeue operation.

The enqueue operation first takes a ticket by incrementing the value of pushTicket with, FAA, which atomically increments the ticket and leaves enqueue with a ticket \( t \). This ticket gives an index, \( t \mod q \), in the array for a single-element queue. Then, enqueue writes an element into the single-element queue by using the turn \( \lfloor t/q \rfloor \). The dequeue operation proceeds in a similar way. It atomically increments popTicket and calculates an index and a turn in the same way. It dequeues a value from the single-element queue and returns this value.
2.4 Relationship to original C++ code

While our high-level implementation of the MPMC queue in ReLoCs ML-like language omits some of the low-level details of the original C++ implementation it remains faithful to the original algorithm. In particular, some of the additional details found in the C++ implementation are:

- The turn sequencer gracefully handles overflow of the turn counter. Our implementation assumes unbounded integers.
- When waiting for a turn, the turn sequencer usconsisting of a with a back-off heuristic consisting of spinning with a back-off and futex’es for increased performance. As standard The MPMC queue concurrent separation logic we model this by only using spinning.
- The MPMC queue supports dynamically increasing the size of the array. However, this feature is deprecated by Facebook (we note that a substantial amount of lines in the C++ implementation serves to support this, but it was later deemed impractical by Facebook).
- It supports additional operations in addition to dequeue and enqueue. For instance, an enqueue operation that fails instead of blocking when the queue is full.

We discuss a few of these details in more depth in Section 10.

3 LINEARIZABILITY OF THE MPMC QUEUE

In this section we analyse the MPMC queue informally and identify its linearization points.

As a first guess, one might think that the linearization point for enqueue is when enqueue writes its value into the single-element queue and, similarly, for dequeue when it reads the value from the single-element queue. After all, these are the points where a value is physically inserted into or read from the data structure. However, placing the linearization points in this way does not work, as the following example shows:

This diagram represents two threads executing operations on the queue. The filled segments represent the duration of the operations. In the example, the first enqueue executes its FAA and receives ticket 0. Afterwards the second enqueue executes its FAA, receives ticket 1, and writes its value to the queue. Then the first enqueue writes its value. Finally a dequeue executes; gets ticket 0, and therefore returns 1. To make this consistent, the linearization point of the first enqueue should happen before the linearization point of the second enqueue. But, the second enqueue writes its value into the queue before the first enqueue does so. Hence, making the linearization points at that time in enqueue is too late.

As the example suggests, the linearization point of the enqueue operations happens at the FAA. If an enqueue operation receives a ticket \(i\), then clearly the value that it insert into the queue is eventually read and returned by the dequeue operation that also receives the ticket \(i\). This means that exactly when the FAA in enqueue is executed, it is determined where in the queue its value is inserted. It thus makes senses to place the linearization point at the FAA. Following this line of argument, we say that the enqueue that receives ticket \(i\) is the \(i\)th enqueue. Moreover, we call the dequeue that receives ticket \(i\) the \(i\)th dequeue, and we say that the \(i\)th enqueue and the \(i\)th dequeue correspond to each other.
It might seem that the linearization point in dequeue is similarly at the FAA operation. This, however, does not always work, as the following example shows:

![Diagram of dequeue and enqueue]

The crux of the example is that dequeue receives ticket 0 before the corresponding enqueue takes its ticket. It is therefore not consistent to put the linearization point of dequeue at its FAA, as dequeue would then take place before the value it returns is enqueued in the first place. However, in general, one can not place the linearization point at when dequeue reads the value either, as that would lead to the same problems as for enqueue.

Thus, the linearization point of the dequeue operation is not always fixed. Looking at the example, we see that we could place the linearization point for the waiting dequeue just after the linearization point for the enqueue that unblocks it. This means that the linearization point of dequeue happens during the execution of enqueue — an external linearization point.

In summary, we conclude the following. If the \( i \)th dequeue arrives after its corresponding enqueue then it has a fixed linearization point at its FAA. If, on the other hand, it arrives before its corresponding enqueue then it has an external linearization point, which happens right after the corresponding enqueue’s linearization point. Observe that even with the external linearization point, it is the case that the \( i \)th dequeue always has its linearization point before the \((i + 1)\)'th dequeue.

**Abstract state.** Given the placement of the linearization points as above, we can talk about the abstract state of the queue, which is determined by the linearized order of the operations. Note that as soon as enqueue receives a ticket, the enqueued element becomes a part of the abstract state, before it is even written into the array. Symmetrically, when a dequeue receives a ticket, it removes an element from the logical queue, even though that value is still present in the physical queue. Thus, the physical state of the underlying array does not determine the abstract state of the queue, e.g., the queue might physically contain no values, while logically it contains arbitrarily many values (and vice versa).

Calculating the abstract state of the queue is important in the refinement proof (Sections 5 and 6), but it is not related directly to the physical state of the array. The abstract state is, however, directly related to the values of \( \text{pushTicket} \) and \( \text{popTicket} \). If \( \text{popTicket} \leq \text{pushTicket} \), then there are exactly \( \text{popTicket} - \text{pushTicket} \) elements in the logical queue. Otherwise the queue is empty and there are \( \text{popTicket} - \text{pushTicket} \) dequeue operations that have arrived before their corresponding enqueue. We will see how these considerations are formalized as part of the refinement proof in Section 6.

### 4 SPECIFICATIONS FOR THE TURN SEQUENCER AND THE SINGLE-ELEMENT QUEUE

In this section we define suitable hoare-triple specifications for the turn sequencer and the single-element queue. We also sketch how these are proved. We emphasize that the proof of the single-element queue only uses the specification (and not the implementation) of the turn sequencer. Similarly, when we prove contextual refinement for the MPMC queue, we only make use of the specification for the single-element queue. Thus our specifications and proofs are modular, and we observe that to prove contextual refinement for the MPMC queue, a unary specification for the single-element queue suffices.
4.1 Turn Sequencer

As mentioned earlier, the turn sequencer is a mechanism for mutual exclusion. Therefore, our specification of the turn sequencer (shown in Figure 3) is an extension of a typical concurrent separation logic specification for a lock [2, 15]. There are two key differences though. The first difference is to enforce that clients of the turn sequencer uses turns correctly. For instance, wait should never be invoked with a past turn. The second difference is that the resource that the turn sequencer protects is indexed with a turn number.

The specification uses two predicates “close” and “isTS”, these are existentially quantified so that they are abstract to clients of the specification (as in [1, 32]). The latter, “isTS”, is the representation predicate. It is persistent which intuitively means that, unlike other separation logic propositions, it is freely duplicable and not consumed by preconditions.

The predicate $R$ describes the resource that the turn sequencer protects. Whereas a lock protects a resource $R: iProp$, the turn sequencer protects a $\mathbb{N}$-indexed family of resources, that is, $R: \mathbb{N} \rightarrow iProp$, where the index represents the current turn. This generalization of the protected resource is possible since the turn sequencer guarantees to run clients in the order of their turns. When it becomes a client’s turn to enter its critical section, it can rely on all earlier turns having been carried out. This allows for “threading” the resource through all the clients, as depicted in the diagram below where the turn sequencer is at the top and its clients at the bottom.

The $R(0)$ in the precondition of newTS ensures that when a turn sequencer is created, the turn sequencer owns the resource for the initial turn. When wait is called with turn $n$, the client receives the resource for that turn, $R(n)$. When completing the turn, the client must give back $R(n + 1)$ and not $R(n)$. This makes it possible for the turn sequencer to give $R(n + 1)$ to next thread in line (which is waiting for the turn $n + 1$).

We now consider the handling of turns in the specification. To represent turns we use ghost state, an Iris feature also found in other separation logics [6, 24, 31]. Ghost state are resources that do not correspond to any physical state of the program. In our case, we want a resource representing
ownership over turns—where owning the turn \( n \) implies that one has the “right” to wait for the \( n \)-th turn. The details of how this ghost state are explained in Section 4.3, but are not important to understand how they are used. The resource \( \text{turns}(\gamma, X) \) denotes ownership over the set of turns \( X \subseteq \mathbb{N} \), and the singular turn \( \text{turns}(\gamma, n) \triangleq \text{turns}(\gamma, \{n\}) \) denotes ownership over a turn \( n \in \mathbb{N} \). These turns can be manipulated, for instance by a client of the turn sequencer, using the rules in Figure 4. The update modality, \( \| \| \), in these rules represents the possibility of updating ghost state and can safely be ignored. The rule \( \text{tokens-alloc} \) states that for any set of natural numbers one can construct a resource for them with a fresh ghost name \( \gamma \). The ghost name can be thought of as a location or variable for the ghost state. Ownership over two disjoint sets of turns, combines into ownership over their union (\( \text{turn-sep} \)). Ownership over two sets of turns that are not disjoint is a contradiction (\( \text{turn-disj} \)). This matches the intuition that two turns can not both have the right to wait for the same turn.

As depicted in the diagram above, when a client creates a new turn sequencer, it acquires ownership over all turns: \( \text{turns}(\gamma, \mathbb{N}) \). Notice how this requires that we can represent an infinite set of turns as ghost state. For readers familiar with Iris, we note that we could instead have used the common approach of using an authoritative resource algebra of finite sets. This would however have made the specification slightly more cumbersome to use as one would then have to allocate new turns by updating the authoritative element. The benefits of our approach are even more evident when we use it to show the specification for the single-element queue. We refer to our Coq formalization for the technical details on how we encode infinite sets using resource algebras.

To call \( \text{wait} \) for a turn \( n \) the client must own \( \text{turns}(\gamma, n) \), the ownership of which is then transferred into the turn sequencer, ensuring that the client can only wait for the same turn once.

Finally, when a client acquires the current turn, it gets \( \text{close}(v, n) \), an exclusive resource giving permission to complete the turn.

**Proof of Specification (Sketch).** To prove that the implementation of the turn sequencer meets the specification, we use the following definitions of the predicates:

\[
\text{close}(\ell, n) \triangleq \ell \xrightarrow{\frac{1}{2}} n
\]

\[
\text{isTS}(\gamma, R, \ell) \triangleq \exists n. \ell \xrightarrow{\frac{1}{2}} n \ast \text{turns}(\gamma, \{m \in \mathbb{N} \mid m < n\}) \ast (R(n) \ast \text{close}(\ell, n) \lor \text{turn}(\gamma, n))
\]

The predicate “\( \text{isTS} \)” is defined as an invariant. An invariant \( \overline{P} \) represents the knowledge that the proposition \( P \) always holds. Since an invariant is knowledge and not a resource that one owns, this definition satisfies the previously mentioned property that “\( \text{isTS} \)” is persistent.

With these definitions we know sketch how the specifications are proved.
For newTS, we have the resource $R(0)$ from the precondition and we obtain $\ell \leftarrow 0$ from stepping through the implementation. We can then allocate the ghost state turns $(\gamma, \mathbb{N})$ using turn-alloc. This allows us to establish the invariant by picking the left disjunct therein.

For wait, we open the invariant around the load. We then have the points-to predicate for the location, and can consider whether the value stored in the location is equal to the turn that wait was called with. In the latter case, we can use induction to handle the recursive call when the check in if fails. In the former case, the turn $(\gamma, n)$ in the right disjunct in the invariant leads to a contradiction, due to the turn $(\gamma, n)$ in the precondition. We thus have the resources in the left disjunct which we can use to show the postcondition, and then close the invariant by showing the right disjunct.

Finally, for complete, we use close$(\ell, n)$ in the precondition to conclude that $n$ is still the current turn, i.e., the existential is equal to $n$. This is the case since close$(\ell, n)$ is in fact half of the points-to predicate for $\ell$. We then have a contradiction in the right disjunct in the disjunction, and symmetrically to what we did for wait, we “flip” the disjunction when we close the invariant.

### 4.2 Single-Element Queue

Similarly to the specification for the turn sequencer, in the specification for the single-element queue (shown in Figure 3) we must ensure that no two dequeue or enqueue operations are performed with the same turn. As such, creating a new single-element queue gives ownership over two sets of turns: one for enqueue and another one for dequeue. These, turns$e(\gamma, n)$ and turns$d(\gamma, n)$, denote ownership over all the turns for enqueue and dequeue, respectively, except for the first $n$ such turns. Additionally, turn$e(\gamma, n)$ and turn$d(\gamma, n)$ represent ownership over the $n$th turn for enqueue and dequeue, respectively. When calling enqueue$SEQ$ or dequeue$SEQ$ with $n$, the specification requires the corresponding turn.

The representation predicate isSEQ is parameterized by a predicate $Q : \mathbb{N} \rightarrow v \rightarrow \text{iProp}$. If $x$ is the $n$th value added to the queue, then $Q(n, x)$ should hold. Correspondingly, the specification for enqueue$SEQ$ requires this in its precondition. This in turn allows the specification for dequeue$SEQ$ to ensure, in its postcondition, that the returned value satisfies the predicate.

**Proof of Specification (Sketch).** To prove that the implementation of the single-element queue meets the specification, we first consider the definition of turn$e$ and turn$d$. These are defined to be ownership over all the even and the odd turns, respectively, except for the first $n$ even or odd numbers:

$$
turns_e(\gamma, n) \triangleq \text{turns}(\gamma, \{m \in \mathbb{N} \mid \text{even}(m) \land 2n \leq m\})$$
$$
turns_d(\gamma, n) \triangleq \text{turns}(\gamma, \{m \in \mathbb{N} \mid \text{odd}(m) \land 2n + 1 \leq m\})$$
$$
turn_e(\gamma, n) \triangleq \text{turn}(\gamma, 2n)$$
$$
turn_d(\gamma, n) \triangleq \text{turn}(\gamma, 2n + 1)$$

Notice how these definitions are only possible because the specification for the underlying turn sequencer allows for ownership over any infinite sets of turns.

Next we define the representation predicate isSEQ:

$$
R_{SEQ}(Q, \ell)(n) \triangleq \begin{cases} 
\ell \leftarrow \text{None} & \text{if even}(n) \\
\exists v. \ell \leftarrow \text{Some } v \cdot Q((n - 1)/2, v) & \text{otherwise}
\end{cases}
$$

$$
isSEQ(\gamma, Q, v) \triangleq \exists ts, \ell. v = (ts, \ell) \cdot \text{isTS}(\gamma, R_{SEQ}(Q, \ell), ts)
$$

It states that the value making up the single-element queue is a pair of a location and a turn sequencer. The representation predicate for the underlying turn sequencer is instantiated with the
resource $R_{SEQ}$, which states that if the current turn is even, then the location points to Non, and otherwise it points to a Some $v$. Since the $n$ given to $R_{SEQ}$ is a turn for the turn sequencer, we must convert it to get a turn for the single-element queue. This is why $R_{SEQ}$ applies $Q$ to $(n-1)/2$.

We can prove the specification for queue$_{SEQ}$ by appealing to the specification for the turn sequencer at its construction. This results in ownership over turns$(\gamma, N)$ which we then split with turn-sep into a set of all even turns (turns$_{(\gamma, 0)}$) and a set of all odd turns (turns$_{(\gamma, 0)}$). The specification for enqueue$_{SEQ}$ and dequeue$_{SEQ}$ are proved simply by using the specification for the turn sequencer. When enqueue$_{SEQ}$ for the $n$th enqueue completes its turn, the turn for the turn sequencer is $2n$, meaning that it must show $R_{SEQ}$ for $2n + 1$, which requires it to show $Q$ for $n$, which is in the precondition. Similar arithmetic show that when enqueue$_{SEQ}$ for $n$ gets its turn, it receives $Q$ for $n$, which it can keep to show the precondition since it only has to show $\ell \rightarrow Non$ when it completes its turn.

### 4.3 Ghost state for turns and tickets

We now detail the construction of the ghost state used to represent turns. This section can be skipped—understanding the derived rules presented in the previous two sections suffices for the rest of the paper.

In Iris ghost state is represented using a form of partial commutative monoids called a resource algebra. The monoid operation $(\cdot)$ combines elements of the resource algebra and a subset of elements $V$ are valid. In the logic the ownership assertion $[a]^\gamma$ denotes ownership over an element $a$ of some resource algebra for a ghost name $\gamma$. Any valid element can be allocated for a fresh ghost name $\gamma$ (Ghost-alloc), ownership of two elements combine into ownership of their combination per the operation (Own-op), and owned elements are always valid (Own-valid).

We want to represent ownership of, potentially, infinite sets of turns. Since ownership of a turn should be exclusive, we want the combination of two sets to be invalid if the sets are not disjoint. The naive approach of letting the elements of the resource algebra be sets and defining the operation as

$$A \cdot B = \begin{cases} A \cup B & \text{if } A \cap B = \emptyset \\ \bot & \text{otherwise} \end{cases}$$

where $\bot$ is invalid, does not work. The operation must be computable, but determining if two arbitrary infinite sets intersect is not.

Instead, we represent sets using a function resembling a characteristic function. We assemble a resource algebra using three standard resource algebras: the exclusive resource algebra, the option resource algebra, and the resource algebra of functions.

$$InfSet(X) = X \rightarrow Option(Ex(1))$$
For the resource algebra of functions, the operation is defined point-wise as \((f \cdot g)(a) = f(a) \cdot g(a)\). Its elements are valid \(f \in \mathcal{V}\) if and only if \(f(a) \in \mathcal{V}\) for all \(a\) in the function’s domain. The codomain \(\text{Option}(\text{Ex}(1))\) has two valid elements \(\text{none}\) and \(\text{some(ex())}\); and one invalid element \(\text{some(⊥)}\). These combine in the following way:

\[
\begin{align*}
\text{none} \cdot \text{some(ex())} &= \text{some(ex())} \\
\text{some(ex())} \cdot \text{some(ex())} &= \text{some(ex()) \cdot ex()} = \text{some(⊥)}
\end{align*}
\]

For any \(A \subseteq X\) we can define an element \(1_A \in \text{InfSet}(X)\) as

\[
1_A(a) = \begin{cases} 
\text{some(ex())} & \text{if } a \in A \\
\text{none} & \text{if } a \notin A
\end{cases}
\]

The idea being that \(1_A\) serves as a sort of characteristic function. Given two disjoint sets \(A\) and \(B\) it is then the case that \(1_A \cdot 1_B = 1_{A \cup B}\). On the other hand, given \(A\) and \(B\) that are not disjoint, then for \(a \in A \cap B\) it is the case that \((1_A \cdot 1_B)(a) = \bot\) and hence the combination is invalid. The three last rules in Figure 5 then follow immediately.

With this in place “turns” is defined simply as

\[
\text{turns}(y, X) \triangleq \prod_{i=1}^{n} x_i^y.
\]

With this definition the previously seen rules for turns in Figure 4 follow from the rules in Figure 5.

5 PROOF OF CONTEXTUAL REFINEMENT

As mentioned, our main result is that the MPMC queue is a contextual refinement of a coarse-grained queue.

**Theorem 5.1.** For all \(q \in \mathbb{N}\) where \(q > 0\),

\[
\vdash \text{queue}_{\text{MPMC}} q \preceq_{\text{ctx}} \text{queue}_{\text{CG}} : \forall \alpha.(1 \to \alpha) \times (\alpha \to 1).
\]

To prove this, we show the following refinement judgment:

\[
\models \text{queue}_{\text{MPMC}} q \preceq \text{queue}_{\text{CG}} : \forall \alpha.(1 \to \alpha) \times (\alpha \to 1).
\]

This reads: the MPMC queue (the implementation) is a contextual refinement of the coarse-grained queue (the specification). The soundness theorem of ReLoC then states that if the above refinement judgment is provable in ReLoC, then the contextual refinement is true in the meta-logic (e.g., Coq or math). The advantage to this approach is that ReLoC offers high-level rules for deriving the refinement judgment. A selection of the rules are given in Figure 6, these are explained as we go along.

As the first step of the proof we observe that the MPMC queue and the coarse-grained queue both consists of a type abstraction (i.e., they are of the form \(\Lambda.e\)) corresponding to he universally quantified type \(\alpha\). Hence we can use the structural rule \(\text{rel-tlam}\). Structural rules use the types and apply when both the implementation and the specification are of the same shape. Specifically, \(\text{rel-tlam}\) states that we must assume that the type \(\alpha\) corresponds to a relation on values \(\tau_i\). Intuitively \(\tau_i\) relates element in the MPMC queue with elements in the coarse-grained queue, it could for instance specify what it means for pairs to be related if \(r\) is a pair type.

Under this assumption we must show that the bodies under the lambdas are related. Since these are not of the same shape, we use symbolic execution rules. These makes it possible to step forward the expression on either side. In the initialization of the MPMC queue we, among other such rules, use \(\text{rel-alloc-1}\) to step over the allocation of the references. At the pair in \(\text{queue}_{\text{CG}}\) we apply the
\[ \Delta \models e_1 \leq t_1 : \tau_1 \quad \Delta \models e_2 \leq t_2 : \tau_2 \]
\[ \Delta \models (e_1, e_2) \leq (t_1, t_2) : \tau_1 \times \tau_2 \]
\[ \forall \tau_1 : \Val \times \Val \rightarrow \text{iProp} \quad \square (\alpha := \tau_1, \Delta \models e_1 \leq e_2 : \tau) \]
\[ \Delta \models \Lambda \models e_1 \leq \Lambda.e_2 : \forall \alpha.\tau \]

\[ P \stackrel{P^N}{\Rightarrow} \Delta \models e_1 \leq e_2 : \tau \]
\[ \Delta \models e_1 \leq e_2 : \tau \]
\[ \forall \nu. (Q(\nu) \Rightarrow \Delta \models K[\nu] \leq e_2 : \tau) \]
\[ I \models K[e_1] \leq e_2 : \tau \]
\[ \forall \ell. \ell \leftarrow \nu \Rightarrow \Delta \models K[\ell] \leq e_2 : \tau \]
\[ \Delta \models K[ref(\nu)] \leq e_2 : \tau \]
\[ \forall \ell \nu. (Q(\nu) \Rightarrow \Delta \models K[\nu] \leq e_2 : \tau) \]
\[ \Delta \models K[\nu] \leq e_2 : \tau \]

\[ \{ \text{True} \} f () \{ \nu. \exists x. R(x, \nu) \} \]
\[ \forall \ell \bar{y}. \left( \| \bar{y} \| = n * \ell \leftarrow \pi_2 \bar{y} * \pi_* \left( R(x, \nu) \right) \right) \Rightarrow \Delta \models K[\ell] \leq e_2 : \tau \]
\[ \Delta \models K[\text{arrayInit } n \ f ] \leq e_2 : \tau \]

**Fig. 6.** ReLoC rules (selection).

\[ \forall w. I_{\text{CG}}(w, x) \Rightarrow \models t \leq K[w] : \tau \]
\[ \models t \leq K[(\text{newlock}(\text{ref}([])))] : \tau \]
\[ I_{\text{CG}}(w, x) \models \models t \leq K[\text{enqueue}_{\text{CG}} w v] : \tau \]
\[ I_{\text{CG}}(w, x) \models \models t \leq K[\text{dequeue}_{\text{CG}} w] : \tau \]
\[ \models \models \text{id } \text{dequeue}_{\text{CG}} w : - \]
\[ \models \models (I_{\text{CG}}(w, x) \models \models t \leq K[w] : \tau) \]

**Fig. 7.** Symbolic execution rules for the coarse-grained queue.

derived rule \text{rel-queue-r} and obtain the representation predicate \( I_{\text{CG}}(w, []) \). After the symbolic execution we have the resources

\[ \ell_{\text{pop}} \leftarrow 0 * \ell_{\text{push}} \leftarrow 0 * \ell_{\text{arr}} \leftarrow \text{map } \pi_2 \text{SEQs } \ast I_{\text{CG}}(w, []) \]  

(1)

where \text{SEQs} is a list of pairs of ghost names and values. We obtain it using \text{array-init-l} and the specification for queue_{SEQ} from Figure 3. Since arrayInit calls queue_{SEQ} 9 times and we apply the specification for queue_{SEQ} for each of them, we get 9 values and ghost names, these are what we have in \text{SEQs}. For each element \((\gamma, v)\) of \text{SEQs} we also get the turns resources turn_{\gamma}(\gamma, 0) and
turn₂(γ, 0), as well as the representation predicate isSEQ(γ, Q, v). We describe an appropriate choice for the predicate Q in the next section, where we describe the invariant.

The definition of ICG can be considered abstract, we only need the rules in Figure 7 (the definition can be found in our Coq formalization).

With the resources described above we now have to show:

\[ \alpha := \tau_i \vdash (\lambda v. enqueue \ell_{arr} q \lambda w. \lambda x. dequeue \ell_{arr} q \ell_{pop}) \]
\[ \preceq (\lambda v. enqueue_{CG} w v, \lambda x. dequeue_{CG} w) \]
\[ : (1 \rightarrow \alpha) \times (\alpha \rightarrow 1). \]

At this point we apply the structural rule rel-pair which requires us to show that the two functions refine the coarse-grained counterparts. To show that the functions are related, we use rel-lam and must show that they are related on related arguments. The precondition of that rule is guarded by the persistence modality □ from Iris. Intuitively, □ P states that P always holds, and to prove it one must show P without relying on any non-persistent resources. A proposition P is said to be persistent, if one can derive □ P from P. In rel-lam the persistence modality, means we have to show that the applied functions are always related. This reflects that for two functions to be related they have to be indistinguishable in any context – including a context that calls the functions several times, potentially in parallel.

This means that the resources in formula (1) can not be used directly as they are not persistent. Instead we must use them to establish an invariant. As mentioned, an invariant \( P \)N persistently states that the proposition P always holds. An invariant can be created using rel-inv-alloc. The contents of an invariant can be accessed for the duration of a single execution step of the implementation (we call this opening an invariant). After taking the step, the user has to show that the invariant still holds (we call this closing an invariant). Before closing the invariant, the user can symbolically execute the specification. The number of execution steps for the specification is not restricted; intuitively, this is because we are showing a form of simulation, matching every execution step in the implementation with (possibly) multiple execution steps in the specification².

Thus, we must now use the resources we have to establish an invariant that relates the abstract state of the MPMC queue with that of the coarse-grained queue. Defining a suitable invariant is the most challenging part of the refinement proof, and we explain it in detail in the next section.

We finish the proof in Section 8, where we use the invariant to show that enqueue and dequeue refine their coarse-grained counterparts.

6 INVARIANT FOR REFINEMENT PROOF

The invariant we use is shown in Figure 8. It is non-trivial and is key to the refinement proof so we devote this section to explain its different parts. Overall, the invariant keeps track of the physical state of the queue, ensures that the queue represents a logic-level list of values, manages the turns for all the single-element queues, and handles the external linearization point.

The invariant is parameterized by the interpretation of the type of values stored in the queue \((\tau_i)\), ghost names \((\gamma_1, \gamma_m, \gamma_i)\), the size of the queue \((q)\), the values for the MPMC queue \((\ell_{pop}, \ell_{push}, \ell_{arr}, \text{SEQs})\), and the value for the coarse-grained queue \((w)\).

Notice the headers in Figure 8, we cover each of these parts of the invariant in turn.

Relation to the coarse-grained queue. The invariant states the existence of two lists \(xs_i\) and \(xs_s\). The state of the coarse-grained queue is tied to \(xs_s\) by \(I_{CG}(w, xs_s)\) and \(xs_i\) is tied to the MPMC

²The way that the condition on the number of steps is enforced is rather technical; see [14] for the details.
\[ I(\tau_i, \gamma_1, \gamma_2, \gamma_3, \gamma_4, \ell_{pop}, \ell_{push}, \ell_{arr}, SEQS, w) \triangleq \exists xs_i, xs_s, \text{popTicket}, \text{pushTicket}, m. \]

**Physical state**

- \( \ell_{pop} \leftarrow \text{popTicket} \)
- \( \ell_{push} \leftarrow \text{pushTicket} \)
- \( \ell_{arr} \leftarrow \square \text{map } \pi_2 \text{SEQs} \)

**Ghost list**

- \( \text{list}(m) \)
- \( |m| = \text{pushTicket} \)
- \( \text{drop}(\text{popTicket}, m) = xs_i \)

**Invariants for the single-element queues**

- \( |\text{SEQs}| = q \left( \begin{array}{c} \star \\ I_{\text{SEQ}}(i, \text{SEQ}_s) \end{array} \right) \)

**Handling of external linearization points**

where

- \( \text{pushObl}(i) \triangleq \text{token}^{\ell_i}(i) \lor \left( \exists id, v_i, v_s, \text{idsAt}^{\ell_i}(i, id) \land \text{listAt}^{\ell_i}(i, v_i) \land \text{turnCtx}(\gamma_i, v_i, v_s) \right) \)

**Relation to the coarse-grained queue**

- \( I_{\text{SEQ}}(i, \gamma, q) \triangleq \text{isSEQ}(\gamma, Q(i, q)) \land \text{turnCtx}(\gamma, i, \text{affectingOps}(\text{pushTicket}, q)) \land \text{turnS}_e(\gamma, \text{affectingOps}(\text{pushTicket}, q)) \land \left( \text{affectingOps}(\text{ops}, q) \triangleq \lfloor \text{ops}/q \rfloor + (\text{if } i < \text{ops} \text{ mod } q \text{ then } 1 \text{ else } 0) \right) \land \text{listAt}^{\ell_i}(j, q + i, v) \)

Fig. 8. Invariant for the MPMC queue

Queue by the rest of the invariant. The separating conjunction over the two lists thus states that the abstract states of the two queues are relate at type \( \tau_i \).

**Physical state.** The physical state of the queue is rather simple. The queue consists of three locations and the invariant contains points-to predicates for all three. As the pointer to the array never changes we represent it using the persistent points-to predicate \( \leftarrow_{\square} \) [40].

**Ghost list.** We previously explained how the physical state of the queue reveals very little about the actual values stored in the queue. We use a ghost list to remedy this.

The ghost list contains all values that have been enqueued, in particular this includes values that are no longer present in the queue. Thus, while the physical state does not change when enqueue executes its FAA, the ghost state does. And, since the linearization point of enqueque is when it increments pushTicket, the number of values that have been added to the queue is always exactly pushTicket. Hence, the ghost list is connected with the physical state in part from the requirement that its length is equal to the value of pushTicket.

Ownership of a ghost list corresponding to the logical list \( xs \) is denoted \( \text{list}^{\ell_i}(xs) \). The resource for the ghost list is ephemeral as it changes over time, when new values are enqueued at the end. This is in fact the only way in which the ghost list changes. This makes it possible to include a
Mechanized Verification of Folly Queue

Fig. 9. Rules for the ghost list.

Indeed, the reason for using the ghost list is exactly to be able to obtain this persistent predicate. Had the ghost list instead reflected only the current list of values in the queue, we would not have been able to make any persistent statements about it, as it could change entirely over time. The ghost list satisfies a number of proof rules presented in Figure 9; these rules are sufficient to carry out the proof.

We can now move on to the following part of the invariant:

\[
\text{list}^y(m) \times |m| = \text{pushTicket} \times \text{drop}(\text{popTicket}, m) = x_s
\]

As alluded to above, the ghost list corresponds to a logical list \(m\) and the length of \(m\) is \(\text{pushTicket}\). Moreover, if we remove the first \(\text{popTicket}\) elements from \(m\) (denoted \(\text{drop}(\text{popTicket}, m)\)), then the remaining list is equal to \(x_s\)—the list which represents the abstract state of the queue. This makes sense, since the ghost list contains all values that have been enqueued and we remove exactly those that have also been dequeued. Note that when \(\text{pushTicket} \leq \text{popTicket}\), then the above implies that \(x_s\) is empty.

Invariants for the Single-Element Queues. For each of the \(q\) single-element queues in the array the invariant needs to include the invariant for the single-element queue and to manage its turns.

We need to instantiate the invariant for each single-element queue with the predicate that holds for the values in it. Recall that this resource is parameterized by the value in the queue and its index. We define \(Q\) to state that the value in the single-element queue corresponds to the “right” value in the ghost list:

\[
Q(i)(j, v) \triangleq \text{listAt}^y(jq + i, v)
\]

Here \(q\) is the capacity of the queue and \(i\) is the index of the particular single-element queue. For the \(j\)th element added to this single-element queue we can then calculate the position of this element in the whole queue as \(jq + i\). Thus, going through the ghost list, we connect the abstract state of the queue to the physical state of all the single-element queues.

In addition we must keep track of the turns for each single-element queue. We must calculate these turns based on the current value of \(\text{popTicket}\) and \(\text{pushTicket}\). The affectingOps aids in this. Given the “global” count of an operation (dequeue or enqueue), it calculates how many times the single-element queue in question was affected.

Handling of external linearization points. The part of the invariant for handling the external linearization point is rather intricate. For the \(i\)th pair of operations, either enqueue or dequeue arrives first. In the former case the invariant should allow both enqueue and dequeue to carry out their own linearization point. In the latter case the invariant must facilitate handling of the external linearization point.
To do this we must intuitively encode the following: When dequeue opens the invariant around its FAA it must transfer the requisite resources into the invariant that will allow another thread to carry out its linearization point. Then, when enqueue opens the invariant around its FAA it should be forced to carry out the corresponding dequeue’s linearization point and transfer the result into the invariant. Later, dequeue needs to open the invariant again, conclude that its linearization point has been carried out, and be able to transfer the resources for the executed linearization point out of the invariant.

**Came-first token.** To keep track of which operation came first we use a tokens—created using ghost state very similarly to what we did for turns earlier. The rules for tokens are seen in Figure 10.

The ith dequeue or enqueue that comes first will be able to take the token\(\gamma^t(i)\). Hence, owning token\(\gamma^t(i)\) proves that an operation came before its corresponding counterpart. The invariant owns all the tokens where neither operation has taken a ticket:

\[
tokensFrom^\gamma(i) = \max(\text{popTicket}, \text{pushTicket})
\]

To see how this allows the operation that arrives first to take a ticket, note that when enqueue and dequeue open the invariant around their FAA, they will close the invariant by using pushTicket + 1 and popTicket + 1, respectively, for the existential variable that they introduced. If enqueue comes first then popTicket \(\leq\) pushTicket. Hence \(\max(\text{popTicket}, \text{pushTicket})\) is equal to pushTicket, and only tokensFrom\(\gamma(i)\) is required for closing the invariant and one token can be kept by enqueue per the rule tokens-take. On the other hand, if enqueue is last, then pushTicket < popTicket and

\[
\max(\text{popTicket}, \text{pushTicket}) = \max(\text{popTicket}, \text{pushTicket} + 1).
\]

Thus when closing the invariant, all the tokens are required and none can be kept. For dequeue the situation is symmetric. All in all, this means that this construction ensures that ith operation that comes first can take the ith token.

**Identifier registry.** Concretely for enqueue to carry out it corresponding dequeue’s linearization point means that it should step dequeue’s specification forward. To this end, \(\models - \preceq_{id} e : -\) represents that some thread, identified by id, needs to show that its implementation refines e. This resource is part of the extensions that we make to ReLoC and is explained in greater detail in Section 7. For now it suffices to know that the state of dequeue’s specification is associated with an

---

---
identifier, $id$, and that dequeue needs a way to ensure that enqueue steps precisely the specification with that identifier forward. To support this, the invariant contains a resource that lets the $i$th dequeue register which identifier it has. The rules for this construction are shown in Figure 11. The resource $\text{ids}^{\forall m}(n)$ represents that only the $n$ first dequeue operations might have registered an identifier. The persistent resource $\text{idsAt}^{\forall m}(i, id)$ represents the knowledge that the $i$th dequeue has registered the identifier $id$.

**Pending dequeues.** When $\text{pushTicket} < \text{popTicket}$, there are $\text{popTicket} – \text{pushTicket}$ dequeue operations blocked, waiting for a value to read. These blocked dequeues are exactly those with external linearization points, and when an enqueue comes along, it should carry out the corresponding dequeue’s linearization point. To this end, enqueue needs some resources, which we store in the invariant:

$$\exists i. \text{idsAt}^{\forall m}(i, id) \ast (\models - \prec id \text{ dequeueCG} \ w : -) .$$

This reads: every $i$th dequeue operation, where $\text{pushTicket} \leq i < \text{popTicket}$, has stored some identifier in the identifier registry and we have the corresponding right refinement, which is ready to invoke dequeue on the coarse-grained queue.

**Enqueue obligation.** The final piece in the invariant is

$$\exists i. \text{pushObl}(\gamma_l, y_t, y_m, \text{pushTicket}) .$$

Since enqueue closes the invariant with $\text{pushTicket} + 1$, it must show $\text{pushObl}(\gamma_l, y_t, y_m, \text{pushTicket})$. Hence, one should think of $\text{pushObl}(i)$ as something which enqueue is *obliged to produce* when it takes the $i$th ticket. Since $\text{pushObl}$ is a disjunction, there are two ways for enqueue to meet this obligation. When enqueue comes first, the obligation is trivial: it can take the token token$^{\forall t} (\text{pushTicket})$, and this is exactly the first disjunct. If, on the other hand, enqueue is last, then there is no way to show the first disjunct and the only option is to show the second disjunct. This involves showing all the things that dequeue needs, including handling dequeue’s linearization point.

**Establishing the Invariant.** To proceed the proof we must now establish the invariant. We must allocate the ghost state using the rules $\text{ghost-list-alloc}$, $\text{tokens-alloc}$, and $\text{identifier-alloc}$. As the parameters for the invariant we choose pick those that correspond from Section 5. When showing the invariant for the existentials we of course pick empty lists for $xs_i$ and $xs_s$ and 0 for $\text{popTicket}$ and $\text{pushTicket}$.

### 7 EXTENDING RELOC WITH SUPPORT FOR EXTERNAL LINEARIZATION POINTS

When verifying refinement for a data structure with an external linearization point, the operation with the external linearization point must let another operation “carry out its linearization point”. In the context of ReLoC, where we prove refinements by symbolically executing two expressions, this means that the other operation must symbolically execute the specification of the first operation. Moreover, in the proof of the first operation it must be possible to keep symbolically executing the implementation up to the point where it can be concluded that another thread has stepped its specification forward. To facilitate this, we should be able to split the state of the implementation and the specification into two separate resources, and transfer the specification side to another thread. This, however, is not possible with the proof rules of ReLoC, because the refinement judgment couples the implementation and specification together.
To support reasoning about external linearization points in ReLoC, we therefore extend the logic with additional rules that allows the user to temporarily decouple a refinement judgment, such that the specification part can be symbolically executed independently of the implementation part.

7.1 Additional rules

A selection of the new rules for external linearization points are shown in Figure 12. The rule \textsc{rel-split} states that one can split a refinement judgment into a left refinement $\vdash e_1 \preceq_{id} - : \tau$ and a right refinement $\vdash - \preceq_{id} e_2 : -$, which represent the state of the implementation and the specification, respectively. One can think of the left refinement as a normal refinement judgment where the specification has been detached. When we split a refinement judgment, we naturally want to keep track of the fact that the two split, left and right, refinements originate from the same refinement judgment. We therefore parameterize the split refinement judgments by an identifier $id$, from an opaque set $\text{Id}$ of identifiers. The rule \textsc{rel-combine}, symmetrically to \textsc{rel-split}, allows one to combine two sides of a refinement into a normal refinement.

Note that in both rules, the right refinement $(\vdash - \preceq_{id} e_2 : -)$ appears on the left-hand side of the magic wand $\Rightarrow$, e.g., \textsc{rel-split} states that one may assume the right hand side and should then prove the left hand side. This allows us to treat the right refinement as a resource\footnote{This treatment of the right refinement stems from the “specifications-as-resources” approach of Turon et al. [35].} and, e.g., transfer it between different threads, for instance by putting it inside an invariant as in the previous section.

In addition to these two rules, we add symbolic execution rules to step forward through the left and right refinement judgments separately. For the left refinement, the symbolic execution rules are straightforward adaptations of the regular left-hand side rules; consider, for example, the direct similarity between \textsc{rel-load-l} and \textsc{rel-load-l}:

\[
\begin{align*}
\text{REL-LOAD-L} & \quad \ell \leftrightarrow v & \quad \ell \leftrightarrow v \Rightarrow \Delta \models K[\ell] \preceq_{id} e_2 : \tau \\
\text{REL-LEFT-LOAD} & \quad \ell \leftrightarrow v & \quad (\ell \leftrightarrow v \Rightarrow K[v] \preceq_{id} - : \tau) \\
& & \quad \Rightarrow K[\ell] \preceq_{id} e_2 : \tau
\end{align*}
\]

In practice (that is, in the Coq implementation), we consider a single rule that generalizes both \textsc{rel-left-load} and \textsc{rel-load-l}. This rule applies to both the normal version of the refinement (when the specification side is present) and the left refinement version (when an identifier is present). This approach both avoids a proliferation of rules and makes it possible to state more general lemmas that apply both in the presence and absence of a right-hand side.
By contrast, the symbolic execution rules for the detached right-hand side look different, compared to the standard ReLoC symbolic execution rules. Typically, symbolic execution rules facilitate backdrops reasoning. This is the case for the “standard” ReLoC rules (Figure 6) and rules for the left refinements. However, per the explanation above, the right refinement is a resource one introduces into the context where one want to do forwards reasoning—exactly what the rules support.

To make sense of it, it is worth looking at a typical use case: we are proving a refinement for a function that performs some “helping”. This amounts to having a goal of the form \( \Delta \models e_1 \preceq e_2 : \tau \) and some right refinement (\( \models \preceq_{id} t : \) ) as an assumption. An example of such a situation (in Coq) is illustrated in Figure 13.

In this example you can see the effects of symbolically executing a load operation in one of the right refinements. The figure also demonstrates the typical interface of an interactive proof. The formula below the line \( \dashv \) is the goal, and the formulas above the line are (named) hypothesis. Our tactic for executing the rel-right-load automatically finds the appropriate hypotheses (“H1” and “H2” in our case) and instantiates the evaluation context \( K \) in the rule. It then applies the rule, and replaces the hypothesis “H2” with an updated version.

As for the Coq tactics for the left refinement, they are the same as the symbolic execution tactics for a normal refinement.

### 7.2 Changes to the ReLoC model

In order to implement those rules, we had to make a number of changes to the model of ReLoC (described in [14]). In this subsection we outline those changes: we show how the left and right refinement judgments are defined and how the rules are encoded. The details presented in this subsection are rather technical, and a reader not familiar with Iris can skip it.

Recall, from [14], that the refinement judgment is defined as:

\[
\models e_1 \preceq e_2 : \tau \triangleq \forall j. K. \{ \text{specCtx} \triangleright j \Rightarrow K[e_2] \} e_1 \{ v. \exists v'. j \Rightarrow K[v'] \triangleright [\tau] (v, v') \}
\]

That is, it is a particular Hoare triple for the left-hand side expression \( e_1 \), specifications for which talk about the thread-pool resource \( j \Rightarrow K[e'] \) and an invariant specCtx (the latter can be ignored). These thread-pool resources are part of the ghost thread-pool: the key element in the definition of the model.

In order to obtain a right refinement, we just package this thread-pool resource \( j \Rightarrow K[e'] \) together with the invariant specCtx. The identifier for such a refinement is then a pair of the thread

---

4This interface is provided by MoSel. [25].

5For reasons of clarity, the definitions given here are presented for a simpler case than was given in [27].
id \ j \ and \ the \ evaluation \ context \ K. \ This \ hides \ all \ the \ unnecessary \ details:  
\[ \text{Id} \triangleq \{ j : \text{nat}, K : \text{ctx} \} \]
\[ \models - \preceq_{id} e_2 : - \triangleq \text{specCtx} * id.j \Rightarrow id.K[e_2] \]

Finally, the left refinement judgment is obtained by taking the definition of a normal refinement, and stripping away the information about the right refinement from the precondition in the hoare-triple:  
\[ \models e_1 \preceq_{id} - : \tau \triangleq \{ \text{True} \} e_1 \{ u, \exists v', id.j \Rightarrow id.K[v'] \} \]

8 REFINEMENT OF ENQUEUE AND DEQUEUE

We now complete the proof of refinement by showing that each of the operations refine their coarse-grained counterpart while assuming the invariant. We have already explained many of the ideas of the proof in the context of the invariant. Here we simply sketch the overall structure of the proof, show how we use the specification for the single-element queue and, in particular, how the external linearization point is handled, including how we use the new and generalized rules covered in Section 7.

8.1 Proof of enqueue

For enqueue we must prove the following refinement:
\[ [\alpha := \tau_i] \models \lambda v. \text{enqueue} \ell_{arr} q \ell_{push} v \preceq \lambda v. \text{enqueue}_{CG} w \quad v : \alpha \rightarrow 1. \]

We use \text{rel-lam} and assume \( v_i \) and \( v_s \) where \( \models [\tau_i]_{\Delta}(v_i, v_i) \). We symbolically execute the implementation up to the FAA and open the invariant around it. This is the critical part of the proof. From the invariant we obtain the points-to predicate \( \ell_{push} \leftrightarrow \text{pushTicket} \), and as the value of \( \ell_{push} \) is changed from \text{pushTicket} to \( \text{pushTicket} + 1 \) we must close the invariant with \( \text{pushTicket} + 1 \) as the witness for the existential variable \text{pushTicket}. To see what this requires we must look at all the occurrences of \text{pushTicket} in the invariant. First of all, from the definition of \text{turnCtx} we can take out \text{turn}(y, n) for the enqueue operation of the \( \text{pushTicket} \) th single-element queue with \( n = [\text{pushTicket}/q] \). Then, we have to update the ghost list. We append an element to the ghost list using \text{ghost-list-append}. This gives us the persistent \( \text{listAt}^{\nu}(\text{pushTicket}, v_i) \) and a ghost list that is one element longer. We now carry out enqueue’s linearization point, meaning we symbolically execute the specification side by using \text{rel-enqueue-r}. Afterwards the abstract state of the coarse-grained queue is \( xs_s + [v_s] \). If \( \text{popTicket} \leq \text{pushTicket} \) we can now trivially close the invariant per the discussion in Section 6. Otherwise we must handle the external linearization point by showing the right disjunct of \( \text{pushObl}(\text{pushTicket}) \). Observe first that if \( \text{pushTicket} < \text{popTicket} \), then \( xs_s \) is the empty list since its length is \( \text{pushTicket} - \text{popTicket} = 0 \). Thus abstract state of the coarse-grained queue is thus the singleton \( [v_s] \). From the collection of pending dequeues in the invariant we get
\[ \exists id. \text{idsAt}^{\nu}(\text{pushTicket}, id) \ast (\models - \preceq_{id} \text{dequeue}_{CG} w : -) \].

We know that the specification queue contains exactly the value \( v_s \), so we can symbolically execute the right-hand side \( \text{dequeue}_{CG} w \) and reduce it to the value \( v_s \) by using \text{rel-dequeue-detached}. This suffices for showing \( \text{pushObl}(\text{pushTicket}) \). We close the invariant while keeping the persistent points-to predicate for \( \ell_{arr} \) and the turn. We can now simply step through the rest of the code. At \text{dequeue}_{SEQ} we apply the unary specification for it and we have both a turn for it and the resource to show the predicate for the value we enqueue.
8.2 Proof of dequeue

For dequeue we prove that \( \lambda x \). dequeue \( \ell_{\text{arr}} q \) \( \ell_{\text{pop}} \) refines \( \lambda x \). dequeue_{CG} w. We again symbolically execute the left-hand side and open the invariant around FAA. From the invariant we get the predicate \( \ell_{\text{pop}} \leftrightarrow \text{popTicket} \). Similarly to what happened in enqueue we must close the invariant with \( \text{popTicket} + 1 \). We can now take the turn turn\( \Delta(y, n) \) for the dequeue of the \( (\text{popTicket} \mod q) \)th single-element queue with \( n = \lfloor \text{popTicket}/q \rfloor \). If \( \text{popTicket} < \text{pushTicket} \), then we can take listAt\( \tau \)(popTicket, \( v \)) using ghost-list-lookup. This corresponds to the first element in the queue and we can use this to step the specification forward using rel-dequeue-\( \tau \).

Otherwise, the linearization point is external, and we apply the rule rel-split and introduce \( (| - \leq_{\text{id}} \text{dequeue}_{CG} w : -) \). We use identifier-decide to get idsAt\( \tau \)(popTicket, \( id \)). We transfer both of these into the invariant when we close it. When we close the invariant we can also keep token\( \tau \)(popTicket) and, as it is persistent, the points-to predicate for \( \ell_{\text{arr}} \). Using the latter we can step over the load of the array. Using the former we can apply the specification for dequeue on the single-element queue. From this specification we know that the dequeue returns a value \( v \) satisfying listAt\( \tau \)(popTicket, \( v \)). We then open the invariant again. Intuitively, at this point an enqueue operation must have carried out dequeue’s linearization point. We can conclude that \( \text{popTicket} < \text{pushTicket} \), since we know that the ghost list has a value at index popTicket, and that the length of the ghost list is pushTicket. Hence the invariant contains pushObl(popTicket); the first disjunct herein being in contradiction with our token\( \tau \)(popTicket). Note how the token we took when we first opened the invariant now serves as a proof that the corresponding enqueue could not have carried out this obligation by providing the token. Hence we have the content in the second disjunct:

\[
\exists id, v, \gamma. \text{idsAt}^{\gamma}(\text{popTicket}, id) \star (| - \leq_{\text{id}} v_s : -) \star (\sigma) \Delta (v, v_s) \star \text{listAt}^{\tau}(\text{popTicket}, v)
\]

We introduce the existentials; by identifier-agree it is clear that the first must be equal to \( id \). We can now apply rel-combine and step the implementation down to the value \( v \) for which we have \( \sigma \Delta (v, v_s) \) and hence these are related.

9 COQ FORMALIZATION

We have formalized all the results mentioned in this paper, building on the formalization of ReLoC and MoSeL. The formalization consists of 1909 lines of Coq code, out of which (a) 366 lines for the implementation and verification of the subcomponents (single-element queue and the turn sequencer); (b) 209 lines for the implementation of the infinite sets resource algebra; (c) 1061 lines for the actual refinement proof. In addition, the formalization includes applications of our methodology to two other examples of algorithms with external linearization points: a version of the elimination-backoff stack from [17], and the red flags versus blue flags example from [36]. The formalization also contains the implementation and verification of a ticket-based lock [29] using the turn sequencer module.

10 DISCUSSION: CONCLUSION, RELATED AND FUTURE WORK

We now discuss related and future work along two dimensions: (1) specification and verification of the MPMC queue, and (2) the extension of ReLoC with support for reasoning about external linearization points.

Wrt. (1), to the best of our knowledge, ours is the first formal specification and verification of the MPMC queue. We emphasize that our verification approach is modular. For example, our specification for the turn sequencer can also be used to verify other clients than the single-element
queue; indeed, in our Coq formalization we have used the turn sequencer to implement and verify a ticket lock. As mentioned, our implementation of the MPMC queue does abstract over some implementation aspects of the Facebook C++ implementation. In particular, the latter makes use of so-called futex’es; whereas in our implementation we use CAS loops when threads have to wait. Futex’es are a mechanism provided by the operating system, which are more efficient than using CAS loops. Futex’es are not modeled by any existing concurrent separation logic; future work thus includes extending our operational semantics model to also account for futex’es and to extend Iris and ReLoC with support for reasoning about futex’es. Such an operational semantics should model that threads, when using a futex, are put to sleep and do not consume system resources in this state.

While it would be interesting to verify the MPMC queue for a weak memory model, we do not think relaxed memory affects the algorithm significantly. Note that accessing the content of the single-element-queues is sequentialized with the turn sequencers; hence there are no races on their content. Additionally, the turn sequencers works by a read-modify-write instruction which is synchronized under weak memory.

The original C++ implementation includes several methods in addition to the standard queue operations enqueue and dequeue. In particular, write and read implement variants of enqueue and dequeue that never blocks. If the queue is full (resp. empty) then they simply return a value indicating failure. It would be interesting to verify the queue with these methods. They do not contextually refine a normal coarse-grained queue and can only satisfy some weaker specification. These operations also make use of acquire reads and hence may pose additional challenges in a relaxed memory setting.

Wrt. (2), we emphasize that our extensions to ReLoC are generally applicable for reasoning about external linearization points. Indeed in our Coq formalization we have applied our methodology to two other examples: a version of the elimination-backoff stack from [17], and the red flags versus blue flags example from [36].

The most closely related work not already discussed earlier in the paper is Liang and Feng’s relational logic [28], which can be used to show refinement for fine-grained concurrent algorithms with non-fixed linearization points, including algorithms with external linearization points. In contrast to Liang and Feng’s logic, our extended version of ReLoC supports a more expressive programming language with higher-order functions (we use them to write out the queue constructors as a function returning a pair of closures).

Another program logic that was used for verifying algorithms with external linearization points is CaReSL [35], which also supports a functional programming language with higher-order functions and higher-order state. As mentioned, our approach to handling external linearization points is closely inspired by the “specifications-as-resources” approach of CaReSL.

We choose to base our approach on ReLoC, and not on the aforementioned logics for the following reasons. Firstly, to our knowledge, ReLoC is the only mechanized logic for interactively proving contextual refinements of concurrent data structures. We have taken advantage of this by using the Coq infrastructure to develop the refinement proof for the MPMC queue. Secondly, ReLoC supports extensions in form of custom ghost state. For example, we used this ability to formulate the turns(\(y, X\)) predicate and its associated rule. The predicate and its rules are encoded using the resource algebra infinite sets. It is not immediately clear how that could be incorporate in the approaches of Liang and Feng [28] and Turon et al. [35]. Finally, ReLoC, and our extension to it, makes it possible to reason with higher-level rules. Whereas in [35], the proof theory is left for the future work, and thus the proofs in CaReSL are carried out directly in the model.
As indicated by the additional case studies mentioned above, we believe that our extension to ReLoC is sufficiently general to support mechanized verification of a wide range of fine-grained concurrent algorithms with external linearization points, and we hope it will be used for that in future work.

Recently, a variant of Liang and Feng’s logic has been formalized in Coq by Zou et. al. [41] and it could be interesting to investigate how a proof of the MPMC queue in that setting would compare with our proof in ReLoC.

In addition to (relational) program logics, there are many proposed methods for verifying data structures with external linearization points. We briefly mention some of them; an interested reader is referred to the survey article [9].

For the case of concurrent queues, Chakraborty et al. [5] provides an alternative to explicitly identifying linearization points in the queue algorithm. Instead, they reduce the problem of finding the linearization points to verifying 4 conditions. The conditions are more “local” than the global linearizability condition, but they are still formulated over execution histories. It is not immediately clear how those conditions can be formulated in our setting, but it would be interesting to see if checking those conditions (or adaptations of them) are easier than formulating a simulation-based proof for the MPMC queue.

As another alternative to identifying linearization points, several methods have been proposed that are based on interval reasoning [8, 10]. This approach is based on interval logic, in which all the predicates are parameterized by time intervals. Interval reasoning has been used to verify linearizability of several algorithms with fixed linearization points, as well as the lazy set algorithm [16], which has an external linearization point for one of the operations. Interval reasoning allows for compositional proofs and abstraction-based reasoning, also for algorithms with external linearization points, but requires formulating the semantics of the underlying programming language in terms of intervals, and, to the best of our knowledge, has not been formalized in a proof assistant.

While there is an abundance of methods for verifying or checking linearizability, we should note that the result that we prove is contextual refinement. To the best of our knowledge, linearizability has not even been properly defined for a programming language with features that we consider here (e.g., higher-order functions, higher-order state, fork-based concurrency).

ACKNOWLEDGMENTS
We are grateful to Peter O’Hearn from Facebook who suggested to us that the MPMQ queue would be an interesting challenge to specify and verify. We also thank Nathan Bronson and Maged Michael from Facebook for informing us about the history of the MPMC queue development at Facebook. This research was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation.

REFERENCES


