Applying Formal Verification to Microkernel IPC at Meta

Quentin Carbonneaux
Meta, France

Noam Zilberstein
Meta and Cornell University, USA

Christoph Klee
Meta, USA

Peter W. O'Hearn
Meta and University College London
UK

Francesco Zappa Nardelli
Meta, France

Abstract

We use Iris, an implementation of concurrent separation logic in the Coq proof assistant, to verify two queue data structures used for inter-process communication in an operating system under development. Our motivations are twofold. First, we wish to leverage formal verification to boost confidence in a delicate piece of industrial code that was subject to numerous revisions. Second, we aim to gain information on the cost-benefit tradeoff of applying a state-of-the-art formal verification tool in our industrial setting. On both fronts, our endeavor has been a success. The verification effort proved that the queue algorithms are correct and uncovered four algorithmic simplifications as well as bugs in client code. The simplifications involve the removal of two memory barriers, one atomic load, and one boolean check, all in a performance-sensitive part of the OS. Removing the redundant boolean check revealed unintended uses of uninitialized memory in multiple device drivers, which were fixed. The proof work was completed in person months, not years, by engineers with no prior familiarity with Iris. These findings are spurring further use of verification at Meta.

1 Introduction

In this paper we report on a project to apply software verification to algorithms at Meta. Static program analysis has been developed and used extensively at Meta, but not formal program verification of full functional correctness. We state up front that the goal of the project was engineering impact rather than generation of novel new scientific techniques. Our aim in writing this paper is to convey our findings on applying formal verification in our specific industrial context, including justification for attempting formal verification at all as well as our reasons for trying some approaches and (crucially) not others. We also identify some of the research problems encountered in the process, progress on which could help make it possible for verification technology to spread further into industry. We intend that this paper should complement other reports on applying software verification in industry (e.g., [6, 11, 25]), and hope that such perspectives can provide input both to future research and to further industrial use of program verification.

1.1 Verification target: IPC

Meta is building an operating system to run on a wide variety of embedded devices [1]; ensuring its reliability and correctness is of utmost importance, making it a good candidate for verification. The OS is being built from scratch to meet the stringent performance constraints of upcoming AR/VR devices. Power considerations are especially crucial, and this has led to a design which eschews locks in favour of non-blocking concurrency. The OS is a microkernel, and inter-process communication (IPC) is one of its central parts: the microkernel design is based on OS components as processes which exchange information via IPC. The whole kernel design — including functionality, security, and privacy — fundamentally relies on IPC. This motivated taking IPC as our first target for formal verification.

Processes in the OS exchange information using a collection of circular, nonblocking, multi-producer, multi-consumer queues. The queues are based on a classic ring-buffer design that is generalized to run in a multithreaded environment and provide some special interface detailed in Sections 4 and 6.1. The queues are implemented using lock-free algorithms. This means that, unlike traditional concurrent programming, the core queue data structure is not made thread-safe by wrapping its operations in `Lock`/`Unlock` calls. Instead, it is synchronized using hardware atomic instructions such as compare and swap. Lock-free algorithms have no critical sections and can interleave arbitrarily; reasoning about them is a real challenge. Indeed, the first version of the queues had one bug that manifested in unlikely conditions and could only be corrected by a significant revision of the implementation. Only three months ago, an assertion in the code was discovered to not hold in rare circumstances.

Our verification goal is to model the programmer's intent using elementary mathematical objects, such as lists of values, and to show that the implementation fits this model. The effect of queue operations in a concurrent setting is described in terms of "linearizability": the property we prove implies that each operation "appears to take effect" at a certain point of time [10]. We validated our specifications with engineers and, as a litmus test, used them to verify a plain queue implemented using an OS queue.

1.2 A pivot: algorithm, not code

At first we took the aim of verifying the actual C code of the core algorithms in the IPC implementation. The entire IPC implementation including control plane operations totals about three thousand lines. While the size of the code is not large, it is intricate. Ideally, the proof and spec would be checked into automatic continuous integration (CI), so as to be kept in synch as the code changed. We ended up not going down this route, for reasons we now explain.

Most fundamentally, scaling program verification to evolving codebases is potentially impactful, but is also a problem with unsolved research challenges [6, 24]. A one-off verification effort can be undone by future code changes, unless proofs evolve too. The cost of doing so is one of the key practical limitations of the celebrated seL4 project [18] and other prominent OS verification efforts. The cost of updating a proof on a code change would at best be unknown, we thought, and in any case we were not aware of data from engineering projects on updating proofs for this kind of concurrent code. The problem of verifying non-blocking concurrency in a scalable way and keeping code+proof synched as the former changed seemed beyond the current state of the field. We say more in Section 8.

A second problem is that previous efforts in OS verification have often restricted concurrency, and developed the program in order to be verified. We had a live kernel already being developed to meet performance constraints.

For these reasons we pivoted to verify core *algorithms*, which we expected would change much less frequently, and not the actual evolving code. This had the following benefits.

- We could spend the most precious resource, human brainpower, on a problem where it is most required — non-blocking concurrency. We did not need to use scarce resources on conceptually simpler but time-consuming problems such as proving memory safety for an implementation.
- It opened up the opportunity to deploy a full-power proof assistant which best fits the proof task, without waiting for one to be wired up to source code + CI + the dialect of C we were considering. With Coq proofs we could target unbounded threads and unbounded inputs, and not just bounded subsets as often done in model checking.
- The engineering team was interested in assurance for the core algorithm(s), and had filed patents on them (since granted [16, 17]), so this addressed an existing concern in a direct fashion.

This pivot was crucial for us to achieve impact without first spending vastly more resources.

We have explained our reasoning here not to justify our choices, but to convey to the reader the kind of considerations concerning cost/benefit analysis that led us to Coq verification in our industrial context. In brief, the cost of code proofs in CI seemed high or unknown (for our problem), and focussing scarce human brainpower on comparatively difficult problems (synchronization) in a Coq proof of an algorithm seemed like a good tradeoff. While interactive theorem proving in systems such as Coq is sometimes considered as a costly activity, we were actually led there by cost – Coq proving was not our original target, and it was surprising to us and colleagues in Meta when we ended up there.

1.3 Choosing Iris

Concurrent separation logic (CSL) [23] is a theory for proving shared memory concurrent programs. Several automatic verification tools use CSL, and a number of interactive proof tools built on top of Coq. Iris [14, 15], one of the latter variety, was a natural choice for the following reasons.

- Prior Research: non-blocking algorithms had previously been verified in Iris [13, 28].
- Corroboration: Iris is not only used by its inventors but also by other research teams (e.g., [2, 8, 20]) and by at least one company, Bedrock Systems.
- Community: Iris has a responsive support community.

Because of Prior Research, we were confident that Iris could do what we wanted in terms of proof, we were just not sure how much it would cost in the hands of non-insiders or what the ripple effects might be in terms of side impact. Because of Corroboration we expected that many hurdles to use had been overcome; in our experience, such hurdles might even be effectively insurmountable for promising research prototypes untested by others. Because of Community, we knew where to ask if we needed help.

We didn't attempt an exhaustive comparison establishing that Iris is the only or definitely best existing tool that would fit our needs. Rather, these reasons made us confident that Iris would be a choice that would let us get started making steps towards our aim of engineering impact, without falling at obvious technical and usability hurdles.

1.4 Results

We verified simplified descriptions of two queue algorithms expressed in Iris' HeapLang. The generic queue is used by profiling and tracing systems where a kernel component needs to stream information to a user process. It is also used in the block, GPU, USB, and AR device drivers that need to communicate with their associated system service. In addition to being non-blocking, the generic queue also operates in a two-phase manner for both enqueue and dequeue operations. The API allows programmers to first claim a slot and then read or write the data directly in the queue's internal representation. A one-phase enqueue/dequeue can be simulated by combining operations.

The ports queue is a used in the kernel as a lightweight message bus to notify a user process about an event. The main feature of the ports queue is that it allows *reservations*.

Reservations guarantee that a later enqueue will not fail because the queue is full. Unlike claiming a slot for a two-phase enqueue, taking a reservation can be done long before using it, with no risk of stalling the queue. The price to pay for the reservation mechanism is that dequeues are constrained to happen in one shot in the ports queue.

Designing the proof invariant made it clear that in the dequeue operation of the generic queue, one atomic load and one check were redundant and not helping synchronization. We decided to simplify the implementation accordingly, but after doing so the CI of the OS reported test failures. Further investigation revealed that the CI failures were due to the use of an improperly initialized circular queue in at least three device drivers. So while the verification did not find bugs in the queue per se, the simplification it suggested to the algorithm revealed actual bugs in client code. In the future we also hope to use the invariant laid out in the proof to more exhaustively test the production code as part of CI.

1.5 Limitations

We assume a sequentially consistent memory model rather than the actual weak memory model of C. So, further bugs could arise due to reordering of CPU instructions. However, we are not concerned by this class of bugs because the C code does not make use of subtle weak-memory synchronization.

The ports queue supports dynamic growth (added roughly when we started the project). This feature is not part of our model and therefore our psuedocode implementation leaves out some components of the real implementation.

Our specifications do not make any claims about liveness. Concretely, a queue for which all operations would loop endlessly satisfies our specification. This limitation of our work is inherited from Iris' focus on safety.

In the remainder of this paper we describe the queue algorithms, their specifications, and the ideas in the proofs.

2 The generic queue

The first queue that we verified is a non-blocking multi-producer and multi-consumer queue used in the kernel to exchange fixed-size messages between threads. The data structure is a generalization of the classic ring buffer to a concurrent context. The requirement that the queue be non-blocking is what makes this generalization complicated, but also interesting in a kernel context. Indeed, grabbing a lock amounts to blocking a kernel thread for a possibly long amount of time (if the message to transfer is large) and delaying the handling of low-latency requests. Another salient feature of the queue is that it permits enqueue and dequeue operations to happen in two phases. This two-phase mechanism permits to reduce the number of copy operations and helps with overall system performance.

```
1 start_enqueue(q):
2   while true:
3     pc   = atomic_load(q.pc)
4     i, k = pc / q.cap, pc % q.cap
5     ik   = atomic_load(q.itr[k])
6     ok   = atomic_load(q.own[k])
7     if ik == i-1 && ok == PROD :
8       if CAS(q.pc, pc, pc+1) :
9         return (k, &q.dat[k])
```

```
1 mark_ready(q, k):
2   atomic_store(q.own[k], CONS)
3   dmb ish // ARM explicit memory barrier
4   atomic_incr(q.itr[k])
```

```
1 start_dequeue(q):
2   while true:
3     cc   = atomic_load(q.cc)
4     i, k = cc / q.cap, cc % q.cap
5     ok = atomic_load(q.own[k])
6     ik   = atomic_load(q.itr[k])
7     if ik == i  && ok == CONS:
8       if CAS(q.cc, cc, cc+1) :
9         return (k, &q.dat[k])
```

```
1 mark_free(q, k):
2   atomic_store(q.own[k], PROD)
```

Figure 1. Pseudocode for the generic queue

2.1 Implementation

Much like in a classic ring buffer, the queue maintains two indices for reads and writes: the consumer and producer counters. In addition to maintaining an array of fixed-size data cells the queue also uses an owner bitmap and an iteration counter for each entry. The owner bitmap registers if a producer or a consumer currently owns the corresponding entry in the data buffer. The iteration counter was added in a revision of the queue to ensure correctness in extreme wrap-around conditions by remembering the "service cycle" of each data cell. The producer counter, consumer counter, owner bits, and iteration counts are all updated using atomic operations. Initially, the iteration counts are all set to zero, the owner bitmap is filled with `PROD` entries, and both producer and consumer counters are set to the queue capacity.

Figure 1 contains a pseudocode implementation of the enqueue and dequeue paths. To enqueue a message the user first calls `start_enqueue` and obtains an index in the queue as well as a pointer to a queue entry. The user code then writes the value to enqueue at the pointer returned (or produces it in place) and finalizes the enqueue by calling `mark_ready` with the index. Dequeuing is performed similarly using `start_dequeue` then `mark_free`.

In both `start_enqueue` and `start_dequeue`, a retry loop will repeatedly try to claim an element by optimistically performing unsynchronized checks and validating them on lines 8 with an atomic compare and swap operation.

```
1  connect_volume(volume_id):
2    vol = new Volume()
3    create_data_pipeline(&vol.pipeline,
4        on_pipeline_notification)
5    initialize_producer(&vol.submission_queue)
6    initialize_consumer(&vol.completion_queue)
7    request_connection(&vol, volume_id)
8    return vol
9
10 on_pipeline_notification(vol, kind):
11   if kind == INCOMMING_DATA_NOTIFICATION:
12     process_completions(&vol.completion_queue)
```

Figure 2. Simplified excerpt of the OS block device driver showing an erroneous use of the queue and its fix (in gray)

2.2 Code improvements

The generic queue used in production is implemented in C and relies on C11 atomic memory accesses to enforce synchronisation between threads. Additionally, some native ARM barrier instructions have been inserted via inline assembly macros. We manually audited the `dmb ish` memory barrier in the `mark_ready` function (line 3). Despite the difficulty of reasoning on code that mixes C11 atomics and inline assembly (an open research problem), we concluded with the OS engineers that the barrier was redundant and could beneficially be removed. Our justification for this change goes as follows: on the target ARM platform, implementing sequentially consistent atomic writes requires inserting a `dmb ish` instruction before the actual memory write, and, depending on the compilation scheme, also after the actual memory write. Since the explicit ARM barrier is in between, and adjacent to, two atomic sequentially consistent writes, it can safely be removed. We committed this to the production version of the generic queue and to another similar queue not presented here. Albeit this improvement was made without formal justification, we consider it a product of our verification effort because it was prompted by the thorough questioning necessary to abstract the algorithm from the actual implementation.

More interestingly, the code displayed in gray at lines 5 and 7 was part of the C implementation and we formally proved it to be redundant: removing it yields a `start_dequeue` function with the exact same behavior. In contrast, the corresponding owner check in `start_enqueue` on line 7 is critical for correctness. Much to our surprise, removing this provably redundant check from the C implementation triggered failures in the continuous integration of the OS. Upon further investigation, we realized that the failures were in fact due to bugs in multiple users of the queue.

2.3 Erroneous uses

All the bugs revealed by removing the superfluous check in the dequeue path followed the same pattern. In Figure 2 we reproduced the bogus pattern for a block device driver. This driver handles devices providing persistent storage such as solid-state drives and USB flash drives; it is used as a building block for the file system module typically used by application developers. To establish connection with a block device (or "volume"), the client calls a function `connect_volume` with a volume ID. One line 3, this function initializes a kernel-backed data pipeline that will be used by the driver to establish a zero-copy data pipeline between the driver, the device IOMMU and the client. Additionally, the data pipeline provides APIs to allow the driver and client to signal each other to implement control plane operations. The requests and responses travel via two queues residing in shared memory. The submission queue is used to send requests to the block driver and the completion queue is used to return the results. The client is thus the producer on the submission queue and consumer on the completion queue, as witnessed by the two initialization calls on line 5 and 6. Note that unlike in our HeapLang model, the OS API separates the producer and consumer initialization, this is because the two ends of a queue can reside in different address spaces so that one end only gets read access to the other end. An OS queue will only be ready to use when both of its ends are initialized.

The failures observed in CI pointed to the pipeline notification callback `process_completions`. They proved to be caused by dequeue attempts on a completion queue that was half initialized. The mitigation implemented by the driver engineer proved rather simple: instead of merely ignoring the data pipeline notification kind, it had to be checked against a specific value signaling that the driver had sent completion messages (and consequently, had initialized its queue end).

One might wonder why the redundant check prevented the consumer from dequeueing messages. The reason is merely that uninitialized memory was filled with zeroes by the heap allocator and that the `CONS` constant happened to be defined to 1.

3 Reasoning about concurrent objects

In this section we introduce informally the techniques used to reason about concurrent code in Iris.

Specifying stateful imperative code is a well-understood problem [12]. We use pre and postconditions for base statements and compose reasoning using a program logic. Program logics usually represent facts about a piece of code with triples $\{P\}c\{Q\}$ composed of a precondition P, a postcondition Q, and a piece of code c. Their intuitive meaning is that if P holds before starting the command c, then Q holds when the command ends. An example triple could be:

$$\left\{0 \le a < 2^{31}\right\} b = a \ll 1 \left\{b = 2 \times a\right\}$$

The triple states that if a is a non-negative integer less than 2^{31} then the shift left by 1 bit gives a result in b that is twice a.

3.1 Invariants

In a concurrent context, however, things are more complicated. Consider the shift-left triple once again. A racing

thread may decrements the variable a before b gets assigned but after the shift-left operation computes its result. In that case, the meaning of the triple falls apart because we no longer have $b = 2 \times a$ when the statement is done.

One solution to this problem that does not completely forbid shared-state interaction is to move to concurrent separation logic, of which Iris is an implementation. In Iris, the assertions in triples are stable by construction, meaning that they are robust to interference with other threads. Mutations to shared memory are ruled by a set of logical properties called invariants and stored in a global logical store. Invariants can be fetched from the global logical store (opened) in order to modify their contents over atomic instructions. But they need to be returned to the store (closed) right after the atomic instruction has run. The intuition for why this rule is sound is that atomic instructions are uninterruptible and, consequently, the problematic case we observed in the previous paragraph is impossible.

Let us see invariants in action over simple examples. We will consider the invariant $\boxed{x \text{ is even}}$. This invariant allows to prove the following triples:

$$\boxed{x \text{ is even}} \vdash \quad \{\top\}\ y = \texttt{atomic\_load}(x)\ \{y \text{ is even}\}$$

$$\boxed{x \text{ is even}} \vdash \quad \{z \text{ is even}\}\ y = \texttt{atomic\_load}(x);\ \texttt{atomic\_store}(x, y + z)\ \{\top\}$$

In the second triple we assume that the variable z is local to the current thread. Note that because the load and store operations are two distinct atomic statements, the code in this triple is not equivalent to atomically adding z to the shared variable x. It is safe nonetheless because $y + z$ is provably even and the invariant can be restored. In contrast, no pre/postcondition can make this triple derivable:

$$\boxed{x \text{ is even}} \vdash \{?\}\ \texttt{atomic\_incr}(x); \texttt{atomic\_incr}(x)\ \{?\}$$

While it is true that x remains even if it was initially even, the invariant does not hold between the two atomic operations. Invariants are a global resource, so they must hold at all times. In this example, a racing thread may rely on x being even after only the first increment has been performed.

3.2 Logical atomicity

Invariants let us modify shared state in a globally coherent way, but that is not enough to effectively specify a concurrent object in a modular way. To see the problem clearly, let us give a tentative spec for a push operation on a concurrent stack:

$$\{\mathsf{Stk}(s, [x_0, \ldots, x_n])\}\ \mathsf{push}(s, x)\ \{\mathsf{Stk}(s, [x, x_0, \ldots, x_n])\}$$

The Stk predicate links a runtime value s with a mathematical list $[x_0, \ldots, x_n]$ that represents the stack content. Our tentative spec for push simply updates the mathematical list by prepending the argument x to it. The problem with this spec in a concurrent context is that push is not an atomic operation. Consequently, even if we know the stack content right before entering the push function, a racing push or pop may change the stack before it is modified by the local push operation. In lock-free programming (without critical sections), this problem seems to prevent giving modular specifications to concurrent objects.

This inadequacy of traditional Hoare triples is mitigated by introducing a new kind of so-called logically atomic specifications [7, 15]. A logically atomic specification is given by a triple with a pre and postcondition. We use angle brackets $\langle\ \rangle$ to tell them apart from classic Hoare triples. In fact, a logically atomic triple is strictly stronger than a Hoare triple: if $\langle P \rangle c \langle Q \rangle$ is derivable, so is $\{P\}c\{Q\}$. The difference between the two judgements is that a logically atomic triple allows opening invariants. Concretely, the following rule holds:

$$\frac{\langle \triangleright I * P \rangle\ c\ \langle \triangleright I * Q \rangle}{\boxed{I} \vdash \langle P \rangle\ c\ \langle Q \rangle} \quad \text{(LAInv)}$$

This rule expresses that, from the user's point of view, a concurrent object specified using logical atomicity behaves exactly as if it were implemented by a single atomic instruction. More often than not, concurrent objects are implemented with multiple instructions. In this case, the logically atomic triple $\langle P \rangle c \langle Q \rangle$ can be intuitively understood as expressing the existence of a single atomic instruction in the execution of c (a *commit* or *linearization* point) that has pre and postcondition P and Q, respectively.

A logically atomic specification for push would be:

$$\langle \forall (x_i).\ \mathsf{Stk}(s, [x_0, \ldots, x_n]) \rangle\ \mathsf{push}(s, x)\ \langle \mathsf{Stk}(s, [x, x_0, \ldots, x_n]) \rangle$$

Note the presence of a universal quantifier for the values (x_i) in the pre and postcondition. This quantifier is required to express that the push operation will work on any list that happens to be the stack content right before the linearization point. In particular, this list may not be the one representing the stack content when entering the push function.

Much like classic Hoare triples, logically atomic specifications are modular. They provide clean interfaces to concurrent objects that can be used in layers to build software.

4 Specifying the generic queue

We now present and justify the logically atomic specification of the generic queue in Iris. Because the API of this queue permits to enqueue and dequeue values in two steps our specification is a bit more intricate than the ones of queues previously formalized in Iris.

Figure 3 contains the specification of the queue creation function and its four core operations. Unlike the other operations, the creation function does not require a logically atomic specification. That is because creating a concurrent object is not an operation that can interfere with shared state. When the queue is created, two predicates are returned in the postcondition: a *persistent* IsQueue predicate, and an affine QueueContent predicate. Persistent predicates are not tied to any resource, and thus, can be freely duplicated. Thus,

$$\textsc{IsQueuePersistent} : \forall \gamma\ q.\ \mathsf{Persistent}(\mathsf{IsQueue}(\gamma, q))$$

$$\{cap > 0\}\ \texttt{make\_queue}(cap)\ \{\lambda q.\ \exists \gamma.\ \mathsf{IsQueue}(\gamma, q) * \mathsf{QueueContent}(\gamma, [])\}$$

$$\mathsf{IsQueue}(\gamma, q) \mathrel{-\!\!*} \left\{ \langle \forall (s_k).\ \mathsf{QueueContent}(\gamma, [s_0, \ldots, s_n]) \rangle\ \texttt{start\_enqueue}(q)\ \left\langle \lambda v.\ \exists k \ell.\ \ast \left\{ \begin{array}{l} v = (k, \ell) \\ \mathsf{Enqueuing}(\gamma, q, k, \ell) * \ell \mapsto \_ \\ \mathsf{QueueContent}(\gamma, [s_0, \ldots, s_n, \ell]) \end{array} \right. \right\rangle \right\}$$

$$\mathsf{IsQueue}(\gamma, q) \mathrel{-\!\!*} \left\{ \left\langle \forall (s_k).\ \ast \left\{ \begin{array}{l} \mathsf{Enqueuing}(\gamma, q, k, \ell) * \ell \mapsto v \\ \mathsf{QueueContent}(\gamma, [s_0, \ldots, s_n]) \end{array} \right. \right\rangle\ \texttt{mark\_ready}(q, k)\ \left\langle \lambda().\ \exists i.\ \ast \left\{ \begin{array}{l} s_i = \ell \\ \mathsf{QueueContent}(\gamma, [\ldots, s_{i-1}, v, s_{i+1}, \ldots]) \end{array} \right. \right\rangle \right\}$$

$$\mathsf{IsQueue}(\gamma, q) \mathrel{-\!\!*} \left\{ \langle \forall (s_k).\ \mathsf{QueueContent}(\gamma, [s_0, \ldots, s_n]) \rangle\ \texttt{start\_dequeue}(q)\ \left\langle \lambda v.\ \exists k \ell v_0.\ \ast \left\{ \begin{array}{l} v = (k, \ell) * s_0 = v_0 \\ \mathsf{Dequeuing}(\gamma, q, k, \ell) * \ell \mapsto v_0 \\ \mathsf{QueueContent}(\gamma, [s_1, \ldots, s_n]) \end{array} \right. \right\rangle \right\}$$

$$\mathsf{IsQueue}(\gamma, q) \mathrel{-\!\!*} \langle \mathsf{Dequeuing}(\gamma, q, k, \ell) * \ell \mapsto \_ \rangle\ \texttt{mark\_free}(q, k)\ \langle \lambda().\ \top \rangle$$

Figure 3. Specification of the generic queue in Iris

if multiple threads will be using a queue q, the predicate $\mathsf{IsQueue}(\gamma, q)$ can be split to send one copy to each thread. This is unlike the QueueContent predicate which can only exist in one copy. This affine predicate relates the queue's logical name γ with a model of its content: a list of *slots* that can either be locations (ℓ) for pending enqueues, or fully enqueued values (v).

The operation specifications all follow the same pattern: A persistent $\mathsf{IsQueue}(\gamma, q)$ assertion and a logically atomic triple are connected with the wand separation logic connective ($-\!\!*$). This structure requires the user of the queue to prove that the IsQueue predicate holds before calling any of the queue's methods. Note that it is unnecessary for the specs to return the IsQueue predicate since, because of persistence, it can be duplicated at will. The IsQueue predicate asserts basic well-formedness of the queue value q and links it to a "logical name" γ that is shared by all predicates to identify the queue instance logically. Unlike the triple's precondition, the IsQueue predicate is required to hold for q before the call to an operation and not only before its commit point. In all specifications, variables that appear not quantified are implicitly universally quantified.

We will now consider the different queue operations. For `start_enqueue`, the logically atomic triple will modify an arbitrary content of the queue to append a location ℓ existentially quantified in the return value. This location is a pointer to a memory slot inside the queue's internal representation. A permission to modify this memory slot $\ell \mapsto \_$ is also returned and the user is expected to use it to assign the value meant to be enqueued in place. Finally, the postcondition provides a predicate $\mathsf{Enqueuing}(\gamma, q, k, \ell)$, that can be understood as a permission to call `mark_ready` later on.

When a user has written the value to be enqueued at location ℓ they can call the `mark_ready` function to commit the enqueue. If the call succeeds there was an index i such that the original queue content contained location ℓ at position i. The effect of the call to `mark_ready` is to update the queue content so that it contains v at position i. Because concurrent enqueues may have started and even finished, it can be that i is not referring to the last element in the queue content.

To dequeue a value, the user calls `start_dequeue`. From the postcondition we see that this function only succeeds when the first slot in the queue content is a value. Any attempt to dequeue when the first slot is still a pending enqueue will block the thread. Like in `start_enqueue`, the function returns a permission to access a memory location in the internal queue representation, as well as a Dequeuing predicate similar to Enqueuing. The points-to predicate gives the user a way to fetch the value that was in the queue, and the queue content is updated to remove the top value.

Because Iris is an affine logic a user may drop the Dequeuing predicate as well as the points-to permission $\ell \mapsto v_0$ instead of calling `mark_free`. However, in the implementation, doing so would end up stalling the queue. This fact is not apparent in our specifications. While that is a shortcoming, the focus on safety has some advantages. For example, our specification works equally well for bounded and unbounded implementations of the queue: if an enqueue is happening when the queue is full, a bounded implementation may simply loop until one slot becomes free. That is how the code in Figure 1 works, and this also explains why the capacity *cap* argument of the `make_queue` function does not appear in any other specification.

Finally, note that there is a slight asymmetry in our specification: pending enqueues are visible in the queue's logical state but not partial dequeues. Making pending dequeues visible may help when specifying liveness properties of the implementation, but we found no need to do it to make our safety specification usable.

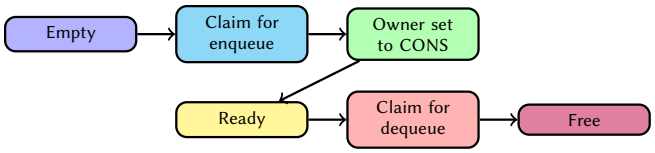


Figure 4. Cell logical states

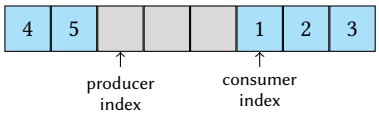
5 Proving the generic queue

We now move to the description of our proof that the generic queue implements the specification in Figure 3. Our proof relies on a global invariant that accounts for all the possible configurations the physical representation of the queue. Leveraging Iris ghost state [15], the invariant is able to express a *protocol* that constrains the evolution of the queue and enables reasoning about sequences of atomic steps.

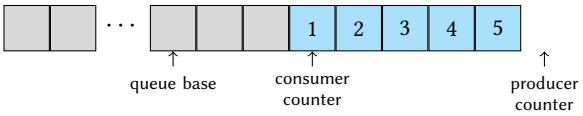
5.1 Endless ribbon

To spare ourselves painful reasoning about modular arithmetic, we architected a logical data structure that does not exhibit the wrap-around behavior of ring buffers. A natural idea that proved very fruitful is to "unroll the ring" into an ever-growing ribbon. This ribbon is indexed with logical indices, as opposed to physical indices that are used to index the various queue arrays. This distinction and two lemmas relating physical and logical indices effectively factored out all the reasoning about modular arithmetic.

The following figure gives an example state for the physical queue; we see that the producer index (i.e., `q.pc` modulo `q.cap`) already wrapped with respect to the consumer index. Both the producer and consumer indices point at the next cell that is going to be produced or consumed.



The corresponding ribbon would be:



In Coq, the logical ribbon is represented as a list of `q.pc` elements. The ribbon items are not mere values, instead they give the *logical state* of the queue cells. Cells are meant to represent one usage cycle of an entry in the physical queue; they start their life in the "Empty" state and end it in the "Free" state. During this cycle a value got enqueued then dequeued. Even though the physical representation of the queue re-uses memory, our logical view of the queue uses a new ribbon cell for each enqueue/dequeue cycle.

To motivate this rich logical state, consider the ribbon depicted above. On the figure, gray cells between the queue base and the consumer counter may still be in the process of being dequeued and have their owner set to CONS because `mark_free` is not yet done. The fine granularity of logical states lets us be explicit about the different configurations in which a queue entry might be. The set of all logical states is displayed in Figure 4. We also used matching colors in Figure 1 to highlight code that transitions to a state of the same color.

5.2 Monotonicity

Monotonicity is central in our reasoning. For example, in the code of `start_enqueue`, the producer counter `q.pc` is only ever increased. Our queue invariant registers this constraint so that every time the producer counter is updated, we have to prove as a side condition that it increased. The counterpart for this proof obligation is that when the compare-and-swap succeeds, we know that the producer counter remained constant during all the atomic loads between lines 3 and 9.

We make another use of monotonicity to constrain the evolution of the cells' logical state. In our invariant, the physical representation of a queue cell — data item, owner bit, and iteration number — is entirely characterized by its logical state. The queue operations are carefully crafted to have this physical representation evolve along a certain order — e.g., the iteration number is always incremented after the owner is set to `CONS`. We encode this protocol as a monotonicity constraint on the evolution of the logical state. Valid transitions between logical states are depicted as arrows in Figure 4.

5.3 Ribbon ghost state

The central resource algebra in our proof is the one for the ribbon ghost state. Figure 5 gives its precise definition as well as derived rules. At a high level, we define three predicates all indexed by a name γ_{ribbon} that ties them to the same ghost resource.

- $\mathsf{Ribbon}(\gamma_{\text{ribbon}}, \boldsymbol{r})$, the authoritative copy of the ribbon's content $\boldsymbol{r}$; the other predicates are fragmental information about specific cells in this "master copy". In our proof, this predicate resides in a global invariant.
- $\mathsf{CellMut}(\gamma_{\text{ribbon}}, i, r)$, a permission granting its owner full access to the ribbon cell at index i. As witnessed by the Ribbon-Agree rule, this permission gives full knowledge of a cell's logical state to its owner.
- $\mathsf{CellBnd}(\gamma_{\text{ribbon}}, i, r)$, a persistent predicate witnessing that the cell at index i is at least in state r. The actual cell content might be a state r' such that $r \rightarrow^+ r'$ (according to the transitions in Figure 4), so r is only a lower bound. The predicate is persistent — holds forever — because cells are constrained to evolve monotonically in Ribbon-Update.

The resource algebra definition is greatly simplified by the various combinators packaged with Iris. We refer the reader to the literature [14, 15] for their precise definition. In any case, the specifics should not obstruct our explanation

$$\textsc{RibbonRA} \triangleq \textsc{Auth}(\textsc{GMap}(\mathbb{N}, \textsc{Auth}(\textsc{Mono}(\text{State}, \rightarrow^*))))$$

$$\text{Ribbon}(\gamma_{\text{ribbon}}, [r_0, \ldots, r_n]) \triangleq \boxed{\bullet\{i \mapsto \bullet r_i \mid 0 \le i \le n\}}^{\gamma_{\text{ribbon}}}$$
$$\text{CellMut}(\gamma_{\text{ribbon}}, i, r) \triangleq \boxed{\circ\{i \mapsto \bullet r\}}^{\gamma_{\text{ribbon}}}$$
$$\text{CellBnd}(\gamma_{\text{ribbon}}, i, r) \triangleq \boxed{\circ\{i \mapsto \circ r\}}^{\gamma_{\text{ribbon}}}$$

(CellBnd-Persistent) $\text{Persistent}(\text{CellBnd}(\gamma_r, i, r))$

(Ribbon-Bound)
$\text{Ribbon}(\gamma_r, \boldsymbol{r}) * \text{CellBnd}(\gamma_r, i, r) \twoheadrightarrow i < |\boldsymbol{r}| \wedge r \rightarrow^* r_i$

(Ribbon-Agree)
$\text{Ribbon}(\gamma_r, \boldsymbol{r}) * \text{CellMut}(\gamma_r, i, r) \twoheadrightarrow i < |\boldsymbol{r}| \wedge r = r_i$

(Ribbon-Bound-Alloc)
$$\frac{\text{Ribbon}(\gamma_r, \boldsymbol{r}) * i < |\boldsymbol{r}|}{\Rightarrow \text{Ribbon}(\gamma_r, \boldsymbol{r}) * \text{CellBnd}(\gamma_r, i, r_i)}$$

(Ribbon-Update)
$$\frac{\text{Ribbon}(\gamma_r, \boldsymbol{r}) * \text{CellMut}(\gamma_r, i, r) * r \rightarrow^* r'}{\Rightarrow \text{Ribbon}(\gamma_r, \boldsymbol{r}[i \leftarrow r']) * \text{CellMut}(\gamma_r, i, r')}$$

Figure 5. Axiomatic presentation of the ribbon ghost state

since we merely use them to realize the axiomatic rules of Figure 5. The only combinator we use that is not part of the Iris standard library is Mono, which comes from Timany & Birkedal [27]. The Mono construction builds a resource algebra from an arbitrary preorder (here $\rightarrow^*$) such that the algebra's inclusion order $\sqsubseteq$ coincides with the preorder. Combined with the the authoritative resource algebra Auth, it can usefully represent the state of single cell. The ribbon is then assembled using a finite map resource algebra GMap and wrapped in another authoritative instance to allow defining both bounds and mutation tokens.

5.4 Invariant

We now turn our attention to the correctness argument. The aim of our reasoning is to show that there are predicate definitions that satisfy the specification in Figure 3. The definitions used in our proofs are presented in Figure 6.

The central predicate $\text{IsQueue}(\gamma, q)$ asserts that q is a well-formed queue instance with logical state identified by the name γ. In our developement, γ is not a single Iris ghost state name but a pair of those $\langle \gamma_r, \gamma_s \rangle$. This is transparent for the user of our specification because γ is treated as a black box. However, it lets us use two ghost structures in other predicates: the logical ribbon (used in Enqueuing) and the user-facing slots list (used in QueueContent). Queue instances are required to be six-tuples with the first item — the queue capacity C — a positive integer and the other items locations for, respectively, the producer and consumer

$$\text{QueueContent}(\gamma, \boldsymbol{s}) \triangleq \boxed{\circ \boldsymbol{s}}^{\gamma_s}$$

$$\text{IsQueue}(\gamma, q) \triangleq$$
$$\exists \gamma_z\, \gamma_c\, C\, \ell_{pc}\, \ell_{cc}\, \ell_d\, \ell_o\, \ell_i.$$
$$C > 0 * q = \langle C, \ell_{pc}, \ell_{cc}, \ell_d, \ell_o, \ell_i \rangle *$$
$$\boxed{\text{QueueInv}(\gamma, \gamma_z, \gamma_c, C, \ell_{pc}, \ell_{cc}, \ell_d, \ell_o, \ell_i))}$$

$$\text{Enqueuing}(\gamma, q, k, \ell) \triangleq$$
$$\exists i. * \begin{cases} i \bmod q.C = k \wedge \ell = q.\ell_d +_l k \\ \text{CellMut}(\gamma_r, i, \text{ClaimedEnq}) \end{cases}$$

(a) Definitions of the specification predicates

$$\text{QueueInv}(\gamma, \gamma_z, \gamma_c, C, \ell_{pc}, \ell_{cc}, \ell_d, \ell_o, \ell_i) \triangleq$$
$$\exists cc\, z\, \boldsymbol{r}.$$
$$* \begin{cases} z \le cc \le |\boldsymbol{r}| = z + C \\ \forall i < z.\ r_i = \text{Free} \\ \forall i < cc.\ \text{Ready}(\_) \rightarrow^+ r_i \\ \forall i \ge cc.\ r_i \rightarrow^* \text{Ready}(\_) \\ \text{MonoNatFull}(\gamma_z, z) * \text{MonoNatFull}(\gamma_c, cc) \\ \text{Ribbon}(\gamma_r, \boldsymbol{r}) * \boxed{\bullet \text{QueueSlots}(\boldsymbol{r}, C, cc, \ell_d)}^{\gamma_s} \\ \ell_{pc} \mapsto z + C * \ell_{cc} \mapsto cc \\ \underset{z \le i < |\boldsymbol{r}|}{*} \begin{cases} r_i = \text{Ready}(\_) \twoheadrightarrow \text{CellMut}(\gamma_r, i, r_i) \\ \text{CellData}(\ell_d +_l (i \bmod C), r_i) \\ \ell_o +_l (i \bmod C) \mapsto \text{CellOwner}(r_i) \\ \ell_i +_l (i \bmod C) \mapsto \text{CellIter}(r_i, i, C) \end{cases} \end{cases}$$

$$\text{QueueSlots}(\boldsymbol{r}, C, cc, \ell_d) \triangleq [\text{Slot}(r_i, i \bmod C, \ell_d) \mid cc \le i]$$
$$\text{Slot}(r_i, k, \ell_d) \triangleq$$
$$\begin{cases} v & \text{when } r_i = \text{Ready}(v) \text{ or } \text{OwnerSet}(v) \\ \ell_d +_l k & \text{otherwise} \end{cases}$$

$$\text{CellData}(\ell, r) \triangleq$$
$$\begin{cases} \ell \mapsto \_ & \text{when } r = \text{Empty} \text{ or } \text{Free} \\ \ell \mapsto v & \text{when } r = \text{Ready}(v) \text{ or } \text{OwnerSet}(v) \\ \text{True} & \text{when } r = \text{ClaimEnq} \text{ or } \text{ClaimDeq} \end{cases}$$
$$\text{CellOwner}(r) \triangleq$$
$$\begin{cases} \texttt{PROD} & \text{when } r = \text{Empty}, \text{ ClaimEnq}, \text{ or } \text{Free} \\ \texttt{CONS} & \text{otherwise} \end{cases}$$
$$\text{CellIter}(r, i, C) \triangleq$$
$$\begin{cases} \lfloor \frac{i}{C} \rfloor - 1 & \text{when } r \rightarrow^* \text{OwnerSet}(\_) \\ \lfloor \frac{i}{C} \rfloor & \text{otherwise} \end{cases}$$

(b) Definition of the internal invariant

Figure 6. Predicates and invariant for the generic queue. In all definitions γ is a pair $\langle \gamma_r, \gamma_s \rangle$ of names: γ_r for the ribbon ghost state, and γ_s for the slots.

counters, the data array, owner array, and iteration array. The IsQueue predicate is persistent since it is an existentially quantified conjunction of persistent predicates. The first two conjuncts are pure and thus persistent. The case of the third conjunct is sorted out by an Iris proof rule showing that every predicate of the form $\boxed{I}$ is persistent.

The definition of the QueueContent predicate asserts fragmental ownership of a list $\boldsymbol{s}$ of queue slots. Iris' ghost state rules then guarantee that the list $\boldsymbol{s}$ is synchronized with the slots described by the authoritative assertion in the QueueInv invariant $\boxed{\bullet\mathsf{QueueSlots}(\boldsymbol{r}, C, cc, \ell_d)}^{\gamma_s}$.

We now turn our attention to the definition of Enqueuing. This predicate is obtained after a user calls `start_enqueue`. It is meant to represent that a slot with identifier k can be set by writing to the heap location ℓ. Our implementation of this predicate requires that there exists a cell in the logical ribbon at index i that is in the state "Claimed for enqueue". The Enqueuing predicate additionally requires that the logical index i is consistent with the physical offset k, and that the location ℓ is at offset k in the queue's data array. We elided the definition of the predicate Dequeuing since it is identical to the one of Enqueuing, except that it requires the state to be "Claimed for dequeue".

Figure 6(b) contains the definition of the global invariant QueueInv. This invariant specifies the entire physical layout of the queue and hinges on three existentially quantified quantities: the consumer counter cc, the queue base z index, and the logical ribbon $\boldsymbol{r}$. The ribbon is split in three segments. The first one, before the queue base z, is dead and all cells there are required to be free. In the second segment $[z, cc)$, cells have been consumed and are either free or in the process of being dequeued; formally their state is required to be strictly greater than Ready in the $\rightarrow$ preorder of Figure 4. Finally, the third segment $[cc, z{+}C)$ contains cells that are being enqueued. Note that only the value of the consumer counter appears directly, the producer counter is computed from the queue base and capacity: $\ell_{pc} \mapsto z + C$. The queue base z and the consumer counter cc are required to evolve monotonically by two MonoNatFull ghost state predicates. The producer and consumer counters in the heap are described with two classic points-to assertions of separation logic. Them being wrapped in the invariant triggers proof obligations when they are mutated: for example, changing cc can only be done if the ribbon cell at index cc can be moved from the enqueuing segment to the dequeuing one. The entry i of the data, owner, and iteration arrays has its content specified by the logical state r_i and the functions CellData, CellOwner, and CellIter, respectively. One interesting thing to notice is that CellData evaluates to True when the logical state is "Claimed for Enqueue/Dequeue". This is because when in these states — i.e., after `start_{enqueue,dequeue}` has completed — the permission to write to (or read from) the data array entry has been lent to the user.

5.5 Code proof

The correctness proofs of the enqueue and dequeue paths are split in two substantially different parts. First we attempt to find out what is the logical state of the cell at the producer/-consumer counter. During this part of the proof we generate persistent CellBnd predicates for each atomic memory read. These predicates and the rule Ribbon-Bound justify the existence of a sequence of monotonically increasing logical states. Coupled with the outcome of the tests on the iteration number and owner bit, this montonic sequence lets us corner a single logical state (Free/Ready). After the compare-and-swap operation, the cell is claimed by the current thread. In practice, this means that we own a CellMut predicate for the cell. This predicate gives us full control over the logical state and lets us update it using Ribbon-Update. When an enqueue operation completes, the CellMut predicate is returned to the invariant so that the matching dequeue operation can fetch it later.

6 The ports queue

The ports queue is used as a lightweight notification system between the kernel and user processes; its implementation, in Figure 7, follows a similar structure to that of the generic queue.

The ports queue offers an original reservation mechanism. The kernel can reserve space in the queue without taking up a physical slot. This guarantees that a later enqueue will not fail due to the queue being full while also allowing other processes to use the queue normally. If instead the kernel were to "reserve" a slot by starting an enqueue without marking the cell ready, then the entire queue would be blocked. That is, no elements could be dequeued until the kernel marked that cell ready. Reservations are used in the kernel to avoid having to cope with error conditions in parts of the code where they are difficult to handle.

The logic for enqueuing is revised to support reservations. The producer counter now tracks three values: the actual counter (`cnt`), the number of existing unused reservations (`res`), and the number of reservations that have been used to enqueue data (`in_flight`). We call *lookahead* the quantity `res+in_flight`. Checking whether a new enqueue or reservation can occur now involves checking whether the space between the producer and consumer counters plus the lookahead is less than the queue capacity.

The price paid for this reservation mechanism is the loss of two-phase dequeues. This modification incurs changes in the synchronization mechanisms, in particular, the owner array present in the generic queue is no longer needed.

Several new operations are added to manipulate reservations. Using a reservation with `use_reservation` is similar to enqueueing a value, but no checks need to be done to ensure that the queue has space. If it is known to the user that a slot enqueued using a reservation has been dequeued,

```
start_enqueue_or_reserve(q, reserve):
  while true:
    pc = atomic_load(q.pc)
    cc = atomic_load(q.cc)
    if (pc.cnt - cc) + pc.res + pc.in_flight < q.cap:
      if reserve:
        newpc = {pc.cnt, pc.res+1, pc.in_flight}
        if CAS(q.pc, pc, newpc):
          return (0, null)
      else:
        newpc = {pc.cnt+1, pc.res, pc.in_flight}
        if CAS(q.cnt, pc, newpc):
          k = pc.cnt % q.cap
          return (k, &q.dat[k])
```

```
use_reservation(q):
  while true:
    pc = atomic_load(q.pc)
    newpc = {pc.cnt+1, pc.res-1, pc.in_flight+1}
    if CAS(q.pc, pc, newpc):
      k = pc.cnt % q.cap
      return (k, &q.dat[k])
```

```
reclaim_reservation(q):
  while true:
    pc = atomic_load(q.pc)
    newpc = {pc.cnt, pc.res+1, pc.in_flight-1}
    if CAS(q.pc, pc, newpc): return
```

```
release_reservation(q):
  while true:
    pc = atomic_load(q.pc)
    newpc = {pc.cnt, pc.res-1, pc.in_flight}
    if CAS(q.pc, pc, newpc): return
```

```
mark_ready(q, k):
  atomic_incr(q.itr[k])
```

```
dequeue(q):
  while true:
    cc   = atomic_load(q.cc)
    i, k = cc / q.cap, cc % q.cap
    ik   = atomic_load(q.itr[k])
    if ik == i:
      data = atomic_load(q.dat[k])
      if CAS(q.cc, cc, cc+1):
        return data
```

Figure 7. Pseudocode for the ports queue

the function `reclaim_reservation` can be used to reclaim the reservation. Finally, reservations can be released using `release_reservation` to free space in the queue.

6.1 Specification

Notice in Figure 7 that operations involving reservations do not validate whether the queue has space. This is because the API assumes that the user does in fact have the reservation that they claim. To specify these functions in Iris, we require that reservations are "passed" to the precondition. Reservations are materialized as purely logical tokens $\mathsf{Res}(\gamma)$ that witness properties about the queue state.

Formal specifications for the most important operations of the ports queue are shown in Figure 8. One notable difference over the generic queue is that slots now optionally bear a marker which we write as superscript R. This marker indicates that the enqueued slot was enqueued using a reservation. When calling dequeue on a queue where the head slot

$$\mathsf{IsQueue}(\gamma, q) \twoheadrightarrow \langle \top \rangle\ \texttt{start\_enqueue\_or\_reserve}(q, \mathsf{true})\ \langle \mathsf{Res}(\gamma) \rangle$$

$$\mathsf{IsQueue}(\gamma, q) \twoheadrightarrow \left\{ \begin{array}{l} \langle \forall (s_k).\ \mathsf{QueueContent}(\gamma, [s_0, \ldots, s_n]) * \mathsf{Res}(\gamma) \rangle \\ \quad \texttt{use\_reservation}(q) \\ \left\langle \lambda v.\ \exists k \ell.\ * \left\{ \begin{array}{l} v = (k, \ell) * \mathsf{Enqueuing}(\gamma, q, k, \ell, \_) \\ \mathsf{QueueContent}(\gamma, [s_0, \ldots, s_n, \ell^{\mathsf{R}}]) \end{array} \right. \right\rangle \end{array} \right.$$

$$\{\mathsf{Enqueuing}(\gamma, q, k, \ell, \_)\}\ \texttt{atomic\_store}(\ell, x)\ \{\mathsf{Enqueuing}(\gamma, q, k, \ell, x)\}$$

$$\mathsf{IsQueue}(\gamma, q) \twoheadrightarrow \left\{ \begin{array}{l} \langle \forall (s_k).\ \mathsf{QueueContent}(\gamma, [s_0, \ldots, s_n]) \rangle \\ \quad \texttt{dequeue}(q) \\ \left\langle \lambda v.\ * \left\{ \begin{array}{l} \exists k.\ s_0 = v^k * (k = \mathsf{R} \twoheadrightarrow \mathsf{DequeuedRes}(\gamma)) \\ \mathsf{QueueContent}(\gamma, [s_1, \ldots, s_n]) \end{array} \right. \right\rangle \end{array} \right.$$

$$\mathsf{IsQueue}(\gamma, q) \twoheadrightarrow \langle \mathsf{DequeuedRes}(\gamma) \rangle\ \texttt{reclaim\_reservation}(q)\ \langle \mathsf{Res}(\gamma) \rangle$$

Figure 8. Selected specifications for the ports queue

is a marked value v^{R}, the dequeuing thread gains access to an instance of the DequeuedRes token. This token can be turned back into a reservation by calling `reclaim_reservation`.

Another subtle difference is that the function starting an enqueue operation no longer returns a bare heap permission $\ell \mapsto \_$. This change is necessary for technical reasons and explained in the following section.

The operations omitted from Figure 8 follow closely their counterpart in the generic queue. For example, starting an enqueue without reservation will append an unmarked location ℓ to the slots list; marking a slot ready turns a location ℓ^k in the slots list into a value v^k with the same marker.

6.2 Code proof

The proof for the ports queue hinges around a similar argument to that of the generic queue. We reuse concepts such as the logical ribbon and monotonicity of the counters and cell states. While cell states still follow the monotonic progression of Figure 4, some states are now skipped.

The intersting part of the proof is to establish that the reservation operations are safe without doing any explicit checks in the algorithm. This is where the reservation resources come into play. As ghost state, we use two instances of the resource algebra $\textsc{AuthNatRA} \triangleq \textsc{Auth}(\mathbb{N}, +)$. This algebra wraps the monoid of natural numbers inside the

authoritative combinator. Some derivable rules are:

(Auth-Nat-Bound) $\boxed{\bullet n}^{\gamma} * \boxed{\circ m}^{\gamma} \vdash m \leq n$

(Auth-Nat-Alloc) $\boxed{\bullet n}^{\gamma} \vdash \Rightarrow \boxed{\bullet (n+m)}^{\gamma} * \boxed{\circ m}^{\gamma}$

The first rule shows that fragments witness lower bounds on the authoritative natural number. The second rule allows allocating a new fragment by increasing the authoritative number. Unlike in a monotonic algebra, this number can also be decreased by consuming fragments.

Two instances of AuthNatRA track the in flight and reservation counters stored in the producer counter. The $\mathsf{Res}(\gamma)$ and $\mathsf{DequeuedRes}(\gamma)$ predicates are simply defined to be $\boxed{\circ 1}^{\gamma_{rs}}$ and $\boxed{\circ 1}^{\gamma_{if}}$, respectively, where γ_{rs} and γ_{if} are two new components of γ. In our reasoning we use these fragments and the Auth-Nat-Bound rule to show that, for example, the lookahead is sufficiently large to justify the presence of one free cell when a reservation is used.

One other subtlety of the ports queue algorithm is that the dequeue function speculatively reads the data array (line 7) before having checked that the cell is in the correct Ready state (line 8). This scheme does not let us return a permission $\ell \mapsto \_$ when we start an enqueue. Indeed, if the permission would be missing from the invariant in some states, we could not possibly prove the safety of the unguarded speculative read on line 7. To deal with this problem, we parameterize Enqueuing with the slot value and prove a Hoare triple for the atomic store expression that assigns to the slot. This way, the user does not need to call a library function to set the slot value but can instead use a regular atomic assignment.

7 Experience using Iris

In this section we report on our experience using Iris for the first time at Meta. Overall, learning and using Iris was smooth. The effort put into the Iris distribution and documentation is remarkable. We made use of the many high-quality resources linked from the Iris website, such as tutorials, related papers, and example developments. We also found the community to be friendly, skilled, and responsive. We discovered the stdpp library at the same time as Iris and found it much more pleasant to use that Coq's standard library, with which we had prior experience. Finally, when designing our invariant we appreciated that the extensive library of resource algebra combinators provided for most of our needs out of the box.

Nonetheless we also experienced some difficulties. HeapLang was well suited to expressing the generic queue algorithm but, when dealing with the compound producer counter of the ports queue, we had to artificially use the Cantor pairing function to pack multiple counters in a single integer. While the C code uses bitpacking, we thought that the modelling language does not ought to restrict the compare-and-swap operation to integers, especially if those are of arbitrary precision. When modelling our invariant we found it difficult to be principled about the split between properties belonging to resource algebras and the ones belonging to the logic. We were also surprised that the most important lemmas took only a couple lines to prove while using the invariants and writing the code proofs required hundreds of rather straightforward lines. While Iris' proof mode [19] made using CSL easy, this observation seems to indicate that there remains untapped potential to increase the reasoning density. Finally, when experimenting with resource algebras on lists, we were surprised by Coq's lack of support for first order reasoning. One of the authors having recently completed a proof in HOL Light repeatedly noticed that some labor intensive reasoning in Coq would have been a one-liner within HOL Light.

8 Contextual remarks

While we stop short of exhaustive comparison to the (large) literature on concurrency and OS verification, in this section we provide remarks on some of the most closely related work, especially that which influenced us.

Fine-grained Concurrency Verification. The GPS paper [28], one of the founding references for Iris, included a circular non-blocking queue as a key example. The queue there was simpler than both our generic and ports queues; e.g., it did not have a reservation mechanism, or two-phase enqueue and dequeue. Also, they proved a certain weak property of the queue which was less than functional correctness (refinement). But, their work provided inspiration for ours.

A recent Iris paper verifies a model of a queue from Meta's Folly C++ library [13]. Going beyond GPS, they prove refinement. The queue algorithm is very different from those in this paper, and used for different purposes. The queue verified is much more complex than the one in the GPS paper; it is used in production at Meta where it serves traffic for their implementation of "the social graph". The realism in this example influenced our choice to proceed with Iris.

Another recent paper on Cosmo [21], which appeared after we had completed our proof, verifies a circular queue which is similar in respects to our generic queue. Both are generalizations of the classic ring buffer that use auxiliary arrays for synchronization. Their queue does not provide a two-phase API nor a reservation mechanism. Nonetheless, the algorithmic similarities yield similarities in the structure of our invariants, such as the central use of monotonicity. One notable difference is is that their reasonning is carried in the Cosmo logic that accounts for the weak memory model of multicore OCaml. Cosmo's treatment of weak memory (or that of GPS) might affect our future work.

The Cosmo authors make a point on the state of the art, which we agree with: *"These logics settle a strong theoretical ground; their confirmation as practical tools, however, needs a demonstration that they allow the modular verification of realistic multicore programs. [21]."* One way to view our work is as providing data on the "confirmation as practical tools"

issue, and since our demonstration is done by engineers who are not the tool creators this perhaps provides additional, more independent data for the community.

Finally, advice from an Iris expert – "for weak memory it is still an open research question what we should be proving, let alone getting engineers to prove it" (Derek Dreyer, personal communication) – influenced our decision to work with a sequentially consistent model for our current proof project, but we look forward to possibly leveraging advances as the field matures.

OS Verification. With demonstrations such as seL4 [18], it is now accepted that it is feasible to prove functional correctness of small OS's, and this had an important influence on us launching our project in the first place. However, our specific industrial problem, paired with the current limitations in OS verification, caused us to focus on proving psuedocode rather than source.

The first issue concerns not doing the proof the first time, but maintaining it once it is done: Our OS is evolving not just in its implementation but also in its requirements. An ideal would be continuous verification, where specs and code are kept in sync using an automatic continuous integration (automatic CI) system that does proofs fully automatically. Less ideal would be "human CI", where human proof experts manually keep specs and code in synch. Currently, an approach like seL4, based on an interactive proof assistant, would tend more towards the human CI end of the spectrum.

A second issue is that the proven systems have often constrained the programming model (e.g., seL4 ruled out preemption), and depending on context such constraints might imperil a non-research, non-demonstrator system.

There has been positive progress (partly) attacking these limitations. Hyperkernel uses a push-button form of verification based on bounded model checking [22], but again at the cost of constraining the programming model. Another work removes the restriction on pre-emption from seL4 [29], but stays closer to what we termed human CI. Work at Google and Meta uses automatic CI for static analysis [9, 26], but for lightweight properties not close to functional correctness. Work at Amazon utilizes CI for proofs of less than functional correctness such as memory safety [3], or for functional correctness of restricted software [5], but to reason about correctness of kernel IPC they moved to a less automatic solution that we understand is not kept in sync using CI [4]. (Side note: the IPC proof of [4] does more than ours in that it considers code and not pseudocode, but less in that it is for coarse-grained rather than fine-grained concurrency.) Finally, the Microsoft F* prover emphasizes automation and CI, but is again driven by proof experts. We hope further progress will help proof to scale more broadly in the future.

9 Conclusion

Formally verifying the generic queue gave us additional benefits on top of the correctness proof. We uncovered improvements to the algorithm as well as two bugs. We are now confident that this piece of the OS functions as intended.

The effort required was less than we initially estimated. The main proof work was done by two engineers with prior experience of Coq but not Iris. Even after we had pivoted to algorithm and not code, we were unsure whether the initial verification could be done inside six months with two engineers. And, we had no way of predicting whether some tangible impact beyond announcing "we're done" would result. In the end it took a person month to come up to speed with Iris, two further person months to verify the Generic queue, and then one further month of part time work for the ports queue. Interaction between Coq and kernel programmers resulted in changes being checked into the kernel. The expertise we built in formal verification of concurrent programs makes us expect that future efforts will be easier.

As mentioned earlier, this was an engineering project and not a project aimed at generating new techniques: it was about experimenting with state-of-the-art formal verification technology in our industrial context, as well as about producing an actual proof. On both fronts, the project was a success. We now know how to use a leading edge tool for verifying concurrent algorithms (Iris), we have produced our first formal proofs using those tools, and demonstrated concretely how side-effect impact via improved code can result. The latter has generated interest within Meta because it shows tangible value beyond confirming that the algorithm is correct. Finally, some generic Coq we have written for this project will be contributed back to the Iris community.

Based on what we have learned, there are numerous follow up projects that we could pursue. There are more components of the IPC stack such as variations of the generic queue which would benefit from formal correctness proofs. The invariants that we used in the formal proof could be useful in other settings as well. These invariants dictate what a valid memory configuration looks like and therefore they could be used in tests to determine whether the actual queue implementation violates the invariants at runtime. While we have proven that the algorithm is correct, additional runtime verification would rule out bugs in the implementation.

We could also bridge the algorithm/implementation gap by making our model more realistic and consider memory models that are not sequentially consistent. This line of work is challenging because verification of concurrent code on top of relaxed memory models is still in its infancy. Nonetheless, building on the invariants of the formal proof, it would be worth investigating if the atomic accesses used by the implementation can be further relaxed for increased performance.

Given the positive outcomes of this project, it makes sense to apply formal verification more widely at Meta.

References

[1] Harry Baker. 2021. Zuckerberg explains why Facebook is building a 'Reality Operating System'. (2021). VR News, 03 June.

[2] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 243–258. https://doi.org/10.1145/3341301.3359632

[3] Nathan Chong, Byron Cook, Jonathan Eidelman, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. 2021. Code-level model checking in the software development workflow at Amazon Web Services. *Softw. Pract. Exp.* 51, 4 (2021), 772–797. https://doi.org/10.1002/spe.2949

[4] Nathan Chong and Bart Jacobs. 2021. Formally Verifying FreeRTOS' Interprocess Communication Mechanism. (2021). Embedded World Exhibition and Conference.

[5] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. 2018. Continuous Formal Verification of Amazon s2n. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II (Lecture Notes in Computer Science)*, Hana Chockler and Georg Weissenbacher (Eds.), Vol. 10982. Springer, 430–446. https://doi.org/10.1007/978-3-319-96142-2_26

[6] Byron Cook. 2018. Formal Reasoning About the Security of Amazon Web Services. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science)*, Hana Chockler and Georg Weissenbacher (Eds.), Vol. 10981. Springer, 38–47. https://doi.org/10.1007/978-3-319-96145-3_3

[7] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–231.

[8] Paulo Emílio de Vilhena and François Pottier. 2021. A separation logic for effect handlers. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434314

[9] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70. https://doi.org/10.1145/3338112

[10] Ivana Filipovi, Peter OHearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for Concurrent Objects. *Theor. Comput. Sci.* 411, 51–52 (Dec. 2010), 4379–4398. https://doi.org/10.1016/j.tcs.2010.09.021

[11] John S. Fitzgerald, Juan Bicarregui, Peter Gorm Larsen, and Jim Woodcock. 2013. Industrial Deployment of Formal Methods: Trends and Challenges. In *Industrial Deployment of System Engineering Methods*, Alexander B. Romanovsky and Martyn Thomas (Eds.). Springer, 123–143. https://doi.org/10.1007/978-3-642-33170-1_10

[12] Robert W. Floyd. 1967. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science (Proceedings of Symposia in Applied Mathematics)*, J. T. Schwartz (Ed.), Vol. 19. American Mathematical Society, Providence, Rhode Island, 19–32.

[13] Dan Frumin, Simon Friis Vindum, and Lars Birkedal. 2021. Mechanized Verification of a Fine-Grained Concurrent Queue from Facebook's Folly Library. (2021). Submitted for publication.

[14] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[15] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

[16] Christoph Klee and Sumit Kamath. 2021. Circular queue for microkernel operating system. (2021). US patent 11113128, Sept 7.

[17] Christoph Klee, Bernhard Poess, and Sumit Kamath. 2020. Port configuration for microkernel operating system. (2020). US patent 10795739, Oct 6.

[18] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. SeL4: Formal Verification of an Operating-System Kernel. *Commun. ACM* 53, 6 (June 2010), 107–115. https://doi.org/10.1145/1743546.1743574

[19] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. https://doi.org/10.1145/3009837.3009855

[20] Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. 2018. Go with the flow: compositional abstractions for concurrent data structures. *Proc. ACM Program. Lang.* 2, POPL (2018), 37:1–37:31. https://doi.org/10.1145/3158125

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

[22] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 252–269. https://doi.org/10.1145/3132747.3132748

[23] Peter W. O'Hearn. 2007. Resources, Concurrency, and Local Reasoning. *Theor. Comput. Sci.* 375, 1–3 (April 2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

[24] Peter W. O'Hearn. 2018. Continuous Reasoning: Scaling the impact of formal methods. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 13–25. https://doi.org/10.1145/3209108.3209109

[25] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella Béguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 983–1002. https://doi.org/10.1109/SP40000.2020.00114

[26] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *Commun. ACM* 61, 4 (2018), 58–66. https://doi.org/10.1145/3188720

[27] Amin Timany and Lars Birkedal. 2021. Reasoning about Monotonicity in Separation Logic. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2021)*. Association for Computing Machinery, New York, NY, USA, 91–104. https://doi.org/10.1145/3437992.3439931

[28] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. *SIGPLAN Not.* 49, 10 (Oct. 2014), 691–707. https://doi.org/10.1145/2714064.2660243

[29] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. 2016. A Practical Verification Framework for Preemptive OS Kernels. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 59–79.