A Verified Messaging System

WILLIAM MANSKY, Princeton University
ANDREW W. APPEL, Princeton University
ALEKSEY NOGIN, HRL Laboratories, LLC

We present a concurrent-read exclusive-write buffer system with strong correctness and security properties. Our motivating application for this system is the distribution of sensor values in a multicomponent vehicle-control system, where some components are unverified and possibly malicious, and other components are vehicle-control-critical and must be verified. Valid participants are guaranteed correct communication (i.e., the writer is always able to write to an unused buffer, and readers always read the most recently published value), while invalid readers or writers cannot compromise the correctness or liveness of valid participants. There is only one writer, all operations are wait-free, and there is no extra process or thread mediating communication. We prove the correctness of the system with valid participants by formally verifying a C implementation of the system in Coq, using the Verified Software Toolchain extended with an atomic exchange operation. The result is the first C-level mechanized verification of a nonblocking communication protocol.

CCS Concepts: • **Software and its engineering** → **Software verification**; *Semantics*; • **Computer systems organization** → *Embedded software*;

ACM Reference format:
William Mansky, Andrew W. Appel, and Aleksey Nogin. 2017. A Verified Messaging System. *Proc. ACM Program. Lang.* 1, 1, Article 1 (October 2017), 28 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In an autonomous vehicle, a variety of sensors must communicate data to a variety of control systems via shared memory. The sensors may take time to write their data, but the control systems must operate in real time, and so can never wait for the next entry to be complete. Instead, each control system should read the most recent complete value from a sensor, while sensors should notify control systems each time a new value is available. If a control system runs much faster than a sensor, it may read the same value several times; if a sensor runs much faster than a control system, some values may be skipped over before they are read. This style of communication is often used in Car Area Network (CAN) systems; the message buffers in such systems are often referred to as "mailboxes".

There are a number of single-writer, multiple-reader systems that might be used to solve this problem, including concurrent-read-exclusive-write (CREW) locks and RCU, a lock-free variant of CREW with extremely lightweight reader synchronization. However, all of these approaches are vulnerable to bad behavior by untrusted components: if a reader refuses to release a lock or acknowledge receipt of new data, it can block the writer from communicating with other readers, and if a writer refuses to release a lock it can block all its readers. If a reader unexpectedly writes to a shared location, it can corrupt existing communication as well. Protection against such buggy or malicious behavior is vital for autonomous vehicles, in which trusted and untrusted subsystems may both read from the same sensors.

To ensure reliable and resilient communication in this scenario, we have developed a messaging protocol in which a small set of data buffers can each contain a sensor value, and dedicated location

2017. 2475-1421/2017/10-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

buffers allow a sensor to publish the location of the newest complete value. A message in a data buffer can be arbitrarily large, and its contents can be read/written using ordinary nonatomic instructions in any order, while the synchronization instructions are single-word and atomic. The protocol is a generic single-writer-multiple-readers system with the following features: all operations are wait-free, readers and writers that follow the protocol are guaranteed to get the most recent communication, and malicious readers or writers cannot compromise the correctness or liveness of the system. The protocol is also zero-copy: once a data buffer is chosen, the writer writes directly to the buffer, and readers read directly from the buffer, without first having to copy the data to a private location.

In this paper, we describe the mechanized formal verification of the correctness of this messaging system, and make semiformal arguments about its liveness. The system is implemented in C and verified with the Verified Software Toolchain (VST) [Appel et al. 2014], using concurrent separation logic (CSL) to reason about the behavior of the C code. Our CSL is based on the work of Hobor et al. [2008], but extended with ghost state in the style of Iris [Jung et al. 2016]. As part of the verification, we construct a CSL specification for an atomic exchange operation, and prove that this specification can be satisfied by a lock-based simulation and can be used in turn to implement a spinlock. All proofs except the semiformal ones of Section 6.2 are formalized in Coq using VST, and can be found online at https://github.com/PrincetonUniversity/VST/tree/master/mailbox.

Our contributions are:

- The design and implementation of a messaging system with useful liveness, security, and efficiency properties.
- A highly general Hoare-logic-style semantics for atomic exchange instructions, exercised by two-sided formal verification (i.e., a verified implementation and a verified use case).
- A formal proof in Coq of the correctness of the C implementation of our messaging system, implying correctness for well-behaved participants. This is the first machine-checked correctness proof of a fine-grained concurrent communication algorithm in real imperative code, and inherits from VST a guarantee that the correctness properties apply to compiled assembly code as well.
- A semiformal argument for the sufficiency of virtual memory protections to prevent ill-behaved participants from interfering with the correctness or liveness of communication.

2 THE MESSAGING SYSTEM

2.1 Scenario

Consider the electronic systems inside a car. Sensors monitor various aspects of the car's state, some safety-critical, some less so. The data from these sensors is transmitted to various control systems, which in turn may be safety-critical or less vital. To provide a strong assurance of the car's safety, we must verify the correctness of all safety-critical components, but may leave complicated and noncritical components unverified. Then we need to know that verified sensors can successfully transmit data to verified control systems, even in the presence of other unverified and potentially buggy or malicious sensors and systems. Writing the data to memory might take several operations, so a sensor should be able to reserve some space in which to write, perform the write, and then make the data available to all readers, and readers should likewise be able to read a message in pieces without it changing partway through.

Reader-writer systems with these requirements are often implemented using reader-writer locks [Courtois et al. 1971]. In such a scheme, each writer is associated with one buffer. Any number of readers can acquire a reader lock on the buffer, allowing them to read the latest data. When the writer wants to update the data, it acquires an exclusive writer lock, which prevents any readers

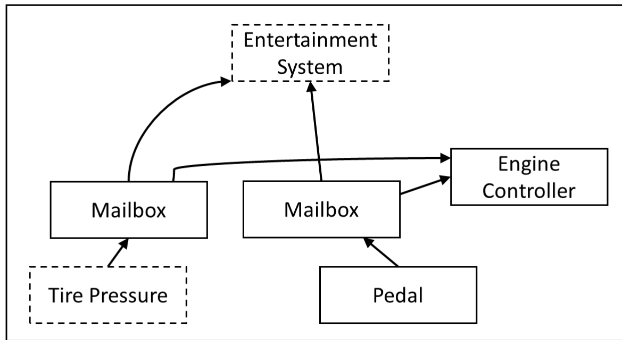

Fig. 1. Car system diagram. Dotted lines indicate unverified, potentially malicious components.

from accessing the buffer, and writes a new value; then it releases the lock and allows readers to read the new value. However, this scheme is vulnerable to malicious components. A malicious reader can hold a lock indefinitely, preventing the writer from acquiring its exclusive lock; a malicious writer can hold the exclusive lock indefinitely, causing all readers to block while waiting for the lock. In either case, a single malicious component can bring all other participants to a halt.

2.2 Design

Our messaging system solves these problems, providing a way for a writer to communicate with multiple readers while guaranteeing that 1) a malicious reader cannot stop a writer from writing or interfere with other readers and 2) a malicious writer cannot stop readers from reading from other writers. Two sets of buffers are used for communication: N location buffers, where N is the number of readers, and $N + 2$ data buffers. All synchronization is performed on the location buffers using atomic exchange instructions, so no party ever blocks, and the location buffer for each reader can be separated from those of all other readers. An atomic exchange instruction, such as the C11 primitive `atomic_exchange`, performs an atomic load-then-store, adding fences as necessary to ensure atomicity[1]. Rather than waiting for all readers to finish reading from a single buffer, the writer maintains enough information to choose a data buffer that is not being read by any reader each time it writes (since there are $N + 2$ data buffers, there will always be at least one that is currently unused). After choosing a data buffer, the writer publishes its choice to the readers via the location buffers, allowing them to switch to reading the newest value. While the atomic exchange instruction is used to communicate the ID of the data buffer to be read, the reads and writes to the data buffers are all ordinary, nonatomic instructions and can be compiled efficiently.

The effects of malicious components are constrained by virtual memory protections: the system is initialized by a trusted registrar that gives each component access to only the necessary memory. The writer process is given read-write access to all associated buffers, but not to any other shared memory (including any other messaging systems that may involve the same readers). Each reader is given read-only access to all buffers, and read-write access to only its own location buffer. This means that if a reader calls `start_read` with any index other than its own, a virtual memory

[1]In our current implementation, we use `atomic_exchange` in sequentially consistent mode. However, we only need enough ordering constraints from `atomic_exchange` to ensure that we can access the data buffers with nonatomic operations. For this reason, it should be possible to relax the memory order to `acq_rel` mode.

protection violation will occur. As described in Section 6.2, this pattern is sufficient to guarantee the desired protections against malicious readers and writers.

```
typedef int buf_id;
#define Empty (-1)
#define B (N + 2) //# of data buffers

buf_id comm[N];
buf_id last_read[N], reading[N];

buf_id start_read(int r) {
   buf_id b =
      atomic_exchange(&comm[r], Empty);
   if (b>=0 && b<B)
       last_read[r] = b;
   else
       b = last_read[r];
   reading[r] = b;
   return b;
}

void finish_read(int r) {
  reading[r] = Empty;
}
```

(a) Reader functions

```
buf_id last_taken[N];
buf_id writing, buf_id last_given;

buf_id start_write(){
  int available[B];
  for (i=0; i<B; i++)
       available[i] = 1;
  available[last_given]=0;
  for (r=0; r<N; r++) {
    if (last_taken[r]!=Empty)
      available[last_taken[r]]=0;
  }
  for (i=0; i<B; i++)
      if (available[i]) {
        writing=i; return i;
      }
  exit(1); //unreachable
}

void finish_write(){
  for (r=0; r<N; r++) {
    buf_id b =
       atomic_exchange(&comm[r], writing);
    if (b==Empty)
       last_taken[r] = last_given;
  }
  last_given = writing;
  writing = Empty;
}
```

(b) Writer functions

Fig. 2. Core functions for the messaging system

The key functions of the messaging system are shown in Figures 2a and 2b. The shared memory consists of the array of per-reader location buffers `comm` and the array of data buffers `bufs`; the latter does not appear in the functions, but all variables of type `buf_id`, including the return values of `start_read` and `start_write`, are interpreted as indices into `bufs`. At the top level, the communication between the writer and reader `r` has the following structure: the writer uses `comm[r]` to publish the index of the most recently written data buffer *b*; the reader responds with `Empty` to acknowledge receipt of the message, then reads from `bufs[b]`; and at some point, the writer receives the acknowledgement and publishes a new most recently written buffer. If the

reader runs more quickly than the writer, it will receive its own acknowledgement message `Empty`, telling it to reuse the last buffer ID it received; if the writer runs more quickly, it will recollect the buffer ID it last sent and publish a new ID instead.

Each reader `r` has local state `last_read[r]`, the most recently received buffer ID, and `reading[r]`, the ID of the buffer currently being read (between a call to `start_read` and a call to `finish_read`, these two values are the same). To start a read, the reader calls `start_read(r)`, which checks `comm[r]` for a new buffer ID (and exchanges it with `Empty` to acknowledge receipt), returns the new ID if one exists (and is in bounds), and returns the previously read ID instead if no new ID has been received (i.e., `comm[r]` contained `Empty`). In either case, the return value of `start_read` is the ID b of a data buffer that reader `r` may now read from, and b is considered to be in use by the reader until `finish_read` is called. When the reader is finished reading, it calls `finish_read(r)` to do some internal bookkeeping, and then it may begin the process again.

The writer's local state consists of `writing`, the ID of the buffer currently being written; `last_given`, the most recently published buffer ID; and `last_taken`, an array that records the last buffer ID that each reader `r` acknowledged receiving. To write a new value, the writer first calls `start_write`, which finds a buffer ID that is not `last_given` and does not appear in `last_taken`. It is an invariant of the system that each active reader `r` is using either `last_taken[r]` or `last_given`, so at most $N + 1$ of the $B = N + 2$ buffers are in use at any time. Thus, the last line of `start_write` is unreachable, and the writer can safely conclude that the buffer ID b returned by `start_write` is not in use by any reader. The writer can then write to data buffer b, using any number of nonatomic write operations. Once the new data is ready to be sent, the writer calls `finish_write`, which publishes b to each reader `r` via `comm[r]` while simultaneously checking whether the reader received and acknowledged the previously published ID `last_given`. If reader `r` acknowledged receiving `last_given` by storing `Empty` in `comm[r]`, the writer updates `last_taken[r]` to `last_given`, recording the fact that the reader is now using `last_given` instead of any buffer it was previously using. If, after this process, a buffer ID no longer appears anywhere in `last_taken`, then no reader is currently using it, and so it is a valid return value for the next `start_write`.

2.3 Comparison with Read-Copy-Update

Read-Copy-Update (RCU) [McKenney and Slingwine 1998] is a popular design pattern for lock-free reader-writer systems, used extensively in the Linux kernel. Our messaging system has some similarities to RCU, but also some crucial differences. In RCU, readers perform synchronizing reads to learn the location of the latest data, and then use nonatomic instructions to read data from that location. When the writer wishes to update data, it copies the current data to a new location in memory, nonatomically updates the new copy, and then atomically publishes the address of the modified data to a shared location. Once the writer knows that no readers are reading the old version of the data, it deallocates the old copy. RCU readers never block, and depending on the implementation perform few to no synchronization operations; the writer, on the other hand, may block while waiting for readers to finish reading the old copy. This makes RCU particularly well suited to scenarios in which reads are much more frequent than writes.

Our messaging system shares some design goals with RCU: readers never have to wait for writers, and both readers and writers can access data with nonatomic instructions once they have learned/chosen its location. Other than that, though, the expectations placed on the readers and writers are different. In our messaging system, the maximum number of readers is known in advance, and we do not assume that reads will be much more frequent than writes. The existence of $N + 2$ data buffers means that at any point in time each reader may be reading a different version of the data, and the writer can still make updates without waiting for readers to catch up.

Many implementations of RCU, on the other hand, allow readers to be out of date by at most one iteration, and in all implementations the writer waits for readers to move on so it can deallocate old copies. Our messaging system has more heavyweight synchronization for readers—`start_read` uses an `atomic_exchange` rather than an atomic load—but the writer's synchronization is fixed at N `atomic_exchanges` rather than depending on the speed of the readers. Our messaging system also allots a separate location buffer to each reader, rather than publishing to a single shared location. This, along with the fact that the writer never blocks, provides much better protection against malicious components: even if a reader decides to write garbage to a synchronization location, it can only interfere with its own communication, and cannot prevent the writer from communicating good data to other readers.

2.4 The Verified Software Toolchain

We prove the correctness of our messaging system for valid participants using the Verified Software Toolchain [Appel et al. 2014]. VST is a Coq-based framework for proving separation logic properties of C programs, along with a guarantee that the properties will still hold on compiled code. Using VST, we can write programs in ordinary C, automatically generate ASTs for them in Coq, and then state and interactively prove specifications for them with some automation support in the form of tactics. The specifications are stated and proved in Verifiable C, a higher-order separation logic for a subset of C, which itself carries a soundness proof relating high-level specifications to the behavior of assembly code produced by the CompCert verified compiler [Leroy 2009]. Verifiable C has recently been extended to include support for Pthreads-style concurrency with locks along the lines of Hobor et al. [2008]. In this paper, we further extend the logic with a proof rule for a C11 atomic operation, but do not prove its soundness; instead, in Section 5, we justify the rule by showing that it is provable on a lock-based simulation of the atomic and can be used to implement correct spinlocks. We then use the extended logic to prove the correctness of the messaging system with valid participants, by proving Verifiable C specifications of the core functions.

3 AXIOMATIZING ATOMIC EXCHANGE

The sequential commands that make up most of the messaging system have well-known separation logic rules; the `atomic_exchange` command, on the other hand, requires further attention. The standard CSL approach to atomic sections of code is to prove tuples in the presence of an environment of resource invariants. Then the atomic execution of a sequence of commands C supports the inference rule [Parkinson et al. 2007]:

$$\frac{\mathsf{emp} \vdash \{P * J\}\ C\ \{Q * J\}}{J \vdash \{P\}\ \mathsf{atomic}\ C\ \{Q\}}$$

where J is the invariant associated with the resource(s) used in C.

In more C-like settings, including the Verified Software Toolchain, we do not have arbitrary atomic blocks; instead, we can perform a fixed set of atomic operations (e.g., lock and unlock) on a certain type of resource (e.g., memory locations that have been designated as locks). The association between a resource and an invariant is then itself part of the triple ("predicates in the heap"), leading to the following reformulation:

$$\frac{\{P * J\}\ C\ \{Q * J\}}{\{P * \mathsf{inv}(r, J)\}\ \mathsf{atomic}\ C\ \{Q * \mathsf{inv}(r, J)\}}$$

where r is the resource used (for instance, r could be the address of a lock, in which case $\mathsf{inv}(r, J)$ is the lock invariant assertion $r \boxdot\!\!\rightarrow J$) and C is an allowed atomic operation on r. In the instance in

which C is a sequence of load and store and r is an atomic location, this could serve as a rule for atomic exchange.

In general, we expect that the invariant J will depend on the current value at the location x. The pre- and postcondition likewise may depend on the values read and written. This leads us to the following final CSL rule for atomic exchange, where the assertion $\mathsf{AEX}(x, J)$ indicates that x is an atomic exchange location with invariant function J:

$$\frac{\forall v_{\text{old}}.\ P(v_{\text{new}}) * J(v_{\text{old}}) \Rightarrow Q(v_{\text{old}}) * J(v_{\text{new}})}{\{P(v_{\text{new}}) * \mathsf{AEX}(x, J)\}\ \mathsf{atomic_exchange}(x, v_{\text{new}})\ \{\lambda v_{\text{old}}.\ Q(v_{\text{old}}) * \mathsf{AEX}(x, J)\}}$$

where the binder $\{\lambda v_{\text{old}}.\ \ldots\}$ in the postcondition binds the return value of the function call. For instance, suppose we have an atomic location x that always contains a pointer to a nonzero value. Then we can choose $J(v) = \exists i \neq 0.\ v \mapsto i$ and use the rule to prove the triple

$$\{p \mapsto i \wedge i \neq 0 * \mathsf{AEX}(x, J)\} y = \mathsf{atomic_exchange}(x, p) \{\exists j \neq 0.\ y \mapsto j * \mathsf{AEX}(x, J)\}$$

in which we exchange the pointer p for the pointer stored in x, maintaining the invariant. As we will see in later sections, this rule also allows more complicated reasoning in which P and J interact to produce Q.

4 GHOST VARIABLES AND HISTORIES

```
          x = 0;
acquire(l); || acquire(l);
x++;        || x++;
release(l); || release(l);
```

Fig. 3. The increment example

Concurrent separation logic, as presented by OHearn [2007], allows us to prove properties of shared-memory concurrent programs, including memory safety and race freedom, by reasoning in terms of ownership of shared locations. However, basic CSL is insufficient to prove correctness of many concurrent programs: it can prove that threads interact safely, but not that they cooperate to accomplish some task. For instance, in the example shown in Figure 3, CSL allows us to prove the postcondition $\mathtt{x} \geq 0$ (a correct invariant for the lock `l`), but not that $\mathtt{x} = 2$ at the end of the program.

To prove the more precise specification, we would like to track the relationship between the work done by each thread and the value of the shared resource `x`. However, once each thread gives up the lock it loses access to the shared resource, and so this relationship cannot be expressed. A common solution, dating back to the earliest work on Hoare-style verification of concurrent programs [Owicki and Gries 1976], is to use *ghost variables* to track relationships maintained between local and shared state. Ghost variables do not appear in the original program, but track information that each thread *could have known* if it had chosen to record that information. When a thread acquires a shared resource, a ghost variable is updated with the latest shared information; when the thread makes a change to the resource, it also modifies the ghost variable to reflect this change. This allows the ghost variable to model "the value last observed" by the thread, acting as a bridge between thread-local and lock-protected information. The increment example of Figure 3 can be straightforwardly verified using ghost variables: each thread is given a ghost variable that starts at 0 and is set to 1 after the increment, and the lock invariant holds that `x` is equal to the sum of the two ghost variables.

In more complex examples, a fixed set of variables may not be sufficient to track the necessary relationships; at the least, it may not be the most natural way to express them. Recent concurrent separation logics such as Iris [Jung et al. 2016] allows general *ghost state* in the form of an arbitrary *partial commutative monoid* (PCM), a set with a partial binary operation (written as $\cdot$) that is associative, commutative, and has a unit. Note that separation logic assertions and the separating conjunction operator $*$ form a PCM, with the empty assertion emp as a unit. Intuitively, any PCM has a "resource-like" structure, in which elements may be combined and rearranged but not removed, and some elements (e.g., two assertions of ownership on the same location) cannot coexist.

We embed arbitrary PCMs into our separation logic with an assertion $a@x$, where a is an element of a PCM and x is a label identifying the particular piece of ghost state. This assertion supports the axiom that $a@x * b@x = (a \cdot b)@x$. An element of ghost state can also be updated via a *frame-preserving update*: we write $a \rightsquigarrow b$ if, for all c, if $a \cdot c$ exists then $b \cdot c$ exists. In other words, ghost state can be changed arbitrarily as long as doing so cannot invalidate any existing assumptions. The frame-preserving update and logical implication are combined into a *view shift* operator $\Rrightarrow$, which is used in an extended rule of consequence:

$$\frac{P \Rrightarrow P' \qquad \{P'\}\ C\ \{Q'\} \qquad Q' \Rrightarrow Q}{\{P\}\ C\ \{Q\}}$$

This allows frame-preserving updates to ghost state as a normal reasoning step in the separation logic.

A particularly expressive form of ghost state is *histories*, which have been used in projects such as VerCors [Blom et al. 2015] and FCSL [Sergey et al. 2015] to verify concurrent data structures. In this approach, the shared resource maintains a list of all the operations performed on it (drawn from a set of events $\mathcal{E}$), which in combination are sufficient to recompute the state of the resource; each client of the resource holds a share of this history, tracking the operations it has performed itself (in FCSL, each client also learns the operations that have been performed by others each time it acquires the resource). Partial histories can be split and combined; when two threads join, each with its own history, the resulting history must take into account the fact that the two partial histories were constructed in parallel.

$$\mathcal{H} = 2^{\mathbb{N}\times\mathcal{E}} \qquad \mathcal{R} = \{u\} + \mathcal{E}^+$$

$$h_1 \cdot h_2 = h_1 \cup h_2 \text{ for disjoint } h_1, h_2 \in \mathcal{H}$$

$$r \cdot u = u \cdot r = r \text{ for } r \in \mathcal{R} \qquad r \cdot r' \text{ undefined otherwise}$$

$$(h_1, r_1) \cdot (h_2, r_2) = (h_1 \cdot h_2, r_1 \cdot r_2) \text{ when compatible}(h_1 \cdot h_2, r_1 \cdot r_2)$$

$$\text{where compatible}(h, r) \triangleq \forall (t, e) \in (h_1 \cdot h_2).\ (r_1 \cdot r_2)[t] = e$$

Fig. 4. The PCM of reference and partial histories

Sergey et al. showed one way to view histories as a PCM: the unit is the empty history, and the operation is disjoint union. In this model, the client carries both a "self" history, describing its own operations, and an "other" history, describing the interference of the environment. This approach is very powerful and has been used to verify work-sharing algorithms such as the flat combiner. However, for programs in which resources are more cleanly separated between threads, a more modular approach is possible, in which clients only hold the histories corresponding to their own operations. Given a shared data structure and a collection of clients, our approach is as follows. First, the data structure itself keeps a record of all operations performed; this *reference* history is

always complete and cannot be split or shared, and contains enough information to reconstruct the state of the data structure. Then, the clients of the data structure also keep *partial* histories, recording the operations each client has performed on the structure; these partial histories can be split, recombined, and passed between threads as virtual resources. A client gains information about the state of the data structure through the guarantee that its partial history is *compatible* with the full reference history. To allow this kind of reasoning, we construct a PCM containing both partial and reference histories over a common set of events $\mathcal{E}$, as shown in Figure 4. A reference history $r \in \mathcal{R}$ is a list of events; a partial history $h \in \mathcal{H}$ is a set of pairs (t, e) of an event and a *timestamp*, which is simply an index into the reference history. The reference PCM $\mathcal{R}$ also includes a special unit element u, and the operation $\cdot$ is only defined when at least one of the operands is u; this ensures that the reference history cannot be split and is always complete. A partial history and a reference history are compatible if each event in the partial history appears in the reference history at the indicated index. Thus, when a client gains access to the reference history (e.g., by acquiring a lock on a shared data structure), it can reason with the knowledge that the events of its local history were actually performed on the shared object in order of timestamp, with other operations by other threads in between.

In order to add an element to the history, we must perform a frame-preserving update on an element that includes the reference history: in particular, $(h, r)@x \rightsquigarrow (h \cup \{(t', e)\}, r + e)@x$, where $+$ is the list append operator, e is some new event, and t' is equal to the length of r. This allows us to add a new event to a partial history and a reference history simultaneously when the client has access to the data structure[2]. Except when a client is accessing the reference history, elements of this PCM will usually have an empty component; we write (h, u) as $\mathsf{hist(h)}$ and $(\emptyset, r)$ as $\mathsf{reference}(r)$.

5 IMPLEMENTING ATOMIC EXCHANGE

In Section 3, we presented the following inference rule for atomic exchange:

$$\frac{\forall v_{\text{old}}.\ P(v_{\text{new}}) * J(v_{\text{old}}) \Rightarrow Q(v_{\text{old}}) * J(v_{\text{new}})}{\{P(v_{\text{new}}) * \mathsf{AEX}(x, J)\}\ \mathsf{atomic_exchange}(x, v_{\text{new}})\ \{\lambda v_{\text{old}}.\ Q(v_{\text{old}}) * \mathsf{AEX}(x, J)\}}$$

In this section, we describe two ways of extending the concurrent separation logic of VST with this rule. First, we simulate the atomic exchange operation with a lock-based implementation, and use existing features of VST to derive the rule; this also gives us a chance to explore the interaction of histories with atomic locations. Next, we provide the rule as an axiom on a built-in atomic exchange operation, and demonstrate its use in building and verifying a spinlock.

5.1 Atomic Exchange via Locks

From a correctness perspective, `atomic_exchange(x, v)` should have the same specification as the following non-atomic code snippet:

```
simulate_atomic_exchange(int *x, lock l, int v){
  acquire(l);
  int w = *x;
  *x = v;
  release(l);
  return w;
}
```

[2]We can also add operations to just the reference history r, but not to the partial history alone, since this would invalidate frames including reference histories that lack the new operation.

where `l` is a lock associated with the atomic location `x`. We can give this code snippet a specification in conventional CSL, but not a useful specification. Once the lock is released, the client loses access to the shared resource, and so gains no useful information from the exchange. As described in the previous section, we can fix this by adding ghost state. In particular, adding a history of operations on the protected value leads to a very general specification for atomic exchange.

The events $\mathcal{E}$ of a history for an atomic exchange location are simply pairs of "value read" and "value written"; we will write $[v_r \to v_w]$ to mean that the value v_r was read and replaced by v_w. An atomic exchange location starts with some value i, and a reference history $r = [v_0 \to v_0'][v_1 \to v_1']...[v_n \to v_n']$ is consistent with i if $v_0 = i$ and $\forall j.\ v_j' = v_{j+1}$. If the history is consistent, the current value at the location is $\mathsf{apply}(i, r) = v_n'$.

We can use this type of history to capture the behavior of an atomic location. An atomic location x has an invariant R, which is a predicate on its full reference history r_x and its current value v_{old}. The client wants to provide some precondition P and receive some useful result Q, predicated on the client's observed partial history h_c and new value v_{new}. Then we say that P, Q, R form a valid atomic exchange triple when the following property holds:

$$\begin{aligned}&\mathsf{AEX_triple}(i, P, Q, R) \triangleq \forall r_x\ h_c\ v_{\text{old}}\ v_{\text{new}}.\ \mathsf{apply}(i, r_x) = v_{\text{old}} \wedge \mathsf{compatible}(h_c, r_x) \Rightarrow \\ &\quad P(h_c, v_{\text{new}}) * R(r_x, v_{\text{old}}) \Rrightarrow Q(h_c \cup \{(|r_x|, [v_{\text{old}} \to v_{\text{new}}])\}, v_{\text{old}}) * R(r_x + [v_{\text{old}} \to v_{\text{new}}], v_{\text{new}})\end{aligned}$$

where $\Rrightarrow$ is the view shift operator described in the previous section and the new timestamp is equal to the length of r_x. The client receives the read value v_{old}, while x receives the written value v_{new}, and both update their histories with the read-write pair (plus the appropriate timestamp in the client). In a given application, the invariant R will be fixed for x, but different clients (or even different parts of the same client) can provide different P's and receive different Q's, as long as they maintain this relationship.

With this idea of valid triple, we can now write a specification for `simulate_atomic_exchange`. We use the assertion $\ell \boxdot\!\!\to R$ to associate a resource invariant R with a lock location ℓ, and $G@p$ to indicate that the ghost state G is associated with the location p. The lock invariant will include the protected location x, the label g of the associated ghost state, the initial value i, the reference history h, and the resource invariant R:

$$\mathsf{AEX_inv}(x, g, i, R) \triangleq \exists r\ v.\ \mathsf{apply}(i, r) = v \wedge x \mapsto v\ * \mathsf{reference}(r)@g * R(r, v)$$

We can then prove the following function specification:

$$\frac{\mathsf{AEX_triple}(i, P, Q, R)}{\begin{array}{l}\{\ell \boxdot\!\!\to \mathsf{AEX_inv}(x, g, i, R) * \mathsf{hist}(h)@g * P(h, v)\}\ \texttt{simulate_atomic_exchange}(x, \ell, v) \\ \{\lambda v'.\ \ell \boxdot\!\!\to \mathsf{AEX_inv}(x, g, i, R) * \exists \mathsf{fresh}\ t'.\ \mathsf{hist}(h \cup \{(t', [v' \to v])\})@g * \\ \ Q(h \cup \{(t', [v' \to v])\}, v')\}\end{array}}$$

The client records the read and write in its own history, and exchanges the precondition on v for the postcondition on v'. Recall from Section 4 that when a thread holds both $\mathsf{hist}(h)@g$ and $\mathsf{reference}(r)@g$ (as it does within the critical section of `simulate_atomic_exchange`), it can combine these into the assertion $(h, r)@g$, and then add a new event to both histories via frame-preserving update. The client does not know that the new timestamp t' is equal to the length of the reference history, only that it is fresh, i.e., later than any timestamp in h.

When verifying programs that use the atomic exchange operation, it is often convenient to think of the client history as part of the state of an atomic exchange location (or, to put it another way, part of the view the current thread has of the atomic location). If we define

$$\mathsf{AEX}(\ell, x, g, i, R, h) \triangleq \ell \boxdot\!\!\to \mathsf{AEX_inv}(x, g, i, R) * \mathsf{hist}(h)@g$$

then the above rule can be written as

$$\frac{\mathsf{AEX_triple}(i, P, Q, R)}{\begin{array}{l}\{\mathsf{AEX}(\ell, x, g, i, R, h) * P(h, v)\}\ \texttt{simulate_atomic_exchange}(x, \ell, v) \\ \{\lambda v'.\ \exists t'.\ \mathsf{AEX}(\ell, x, g, i, R, h \cup \{(t', [v' \to v])\}) * Q(h \cup \{(t', [v' \to v])\}, v')\}\end{array}}$$

This statement of the rule exactly matches the atomic exchange axiom of Section 3, modulo the introduction of histories. Including histories in the specification of atomic exchange allows clients to track the operations they have performed on the shared location and relate the values they observe to previous values of the atomic location, and we will take full advantage of this when verifying the messaging system in Section 6.1.

5.2 Truly Atomic Exchange

If we have access to an atomic exchange instruction in the underlying language (such as the one included in the C11 atomics), then we can apply a similar specification directly to that operation. Rather than using a lock invariant, we add a new basic predicate $x \odot\!\!\rightarrow (g, i, R)$ to indicate that x is an atomic exchange location with ghost label g, initial value i and invariant R. As above, we stretch our notation and write $x \odot\!\!\rightarrow (g, i, R) * \mathsf{hist}(h)@g$ as $x \odot\!\!\rightarrow (g, i, R, h)$. Then we assert the following specification:

$$\frac{\mathsf{AEX_triple}(i, P, Q, R)}{\begin{array}{l}\{x \odot\!\!\rightarrow (g, i, R, h) * P(h, v)\}\ \texttt{atomic_exchange}(x, v) \\ \{\lambda v'.\ \exists \text{fresh } t'.\ x \odot\!\!\rightarrow (g, i, R, h \cup \{(t', [v' \to v])\}) * Q(h \cup \{(t', [v' \to v])\}, v')\}\end{array}}$$

From a verification perspective, the only difference between this and the simulated atomic exchange is that the lock assertion has been replaced by a built-in atomic exchange assertion. In the following proofs, as long as the associated lock can be inferred from context, we will use $x \odot\!\!\rightarrow (g, i, R, h)$ as a predicate that can be implemented by either simulated or axiomatized atomic exchange. From an execution perspective, however, only the axiomatic version provides the nonblocking guarantees we want for the messaging system, and these guarantees are essential to the semiformal reasoning about invalid components in Section 6.2. Our proof of the analogous specification for the simulated version gives us confidence that we have the right specification for atomic exchange.

5.3 Testing the Specification

```
void acquire(x){
  int r = 1;
  while(r)
    r = atomic_exchange(x, 1);
}

void release(x){
  atomic_exchange(x, 0);
}
```

Fig. 5. A simple spinlock implementation

As further evidence that our specification for atomic exchange is the right one, we use it to verify a well-known application of atomic exchange: the construction of spinlocks. A very simple

implementation of spinlocks is shown in Figure 5. The lock at location x is considered to be locked when its value is 1, and unlocked when its value is 0. To acquire the lock, a thread repeatedly exchanges 1 with its current value; if the value read is 0, then the lock has been successfully acquired. To release the lock, a thread atomically writes 0 to the location. This does not force a thread to hold the lock in order to release it, but in order to verify a program using such a lock we will need to show that only threads with the proper permissions call `release`.

The goal, then, is to choose a defined predicate $\mathsf{lock}(\ell, R)$ such that the usual CSL specifications for `acquire` and `release` hold on these functions:

$$\{\mathsf{lock}(\ell, R)\}\ \texttt{acquire}(\ell)\ \{\mathsf{lock}(\ell, R) * R\}$$

$$\{\mathsf{lock}(\ell, R) * R\}\ \texttt{release}(\ell)\ \{\mathsf{lock}(\ell, R)\}$$

We begin by describing an appropriate invariant for the atomic exchange location, given a desired lock invariant R:

$$R_{\text{lock}}(R)(h, v) = \text{if } v = 0 \text{ then } R \text{ else } \mathsf{emp}$$

When the lock is free, the atomic exchange location holds the resource R; otherwise, the lock holder holds the resource, and the atomic exchange location contains no permissions to any memory. Then we can define the lock predicate as:

$$\mathsf{lock}(\ell, g, R) = \exists h.\ \ell \odot\!\!\rightarrow (g, 1, R_{\text{lock}}(R), h)$$

We need no information about past operations in order to reason about the lock, so any arbitrary history is sufficient. We do, however, need to pass the ghost label g as an argument to the predicate, so that it can be split and rejoined without losing track of the label.

Given this definition, and either version of the atomic exchange rule from the previous sections, we can prove the desired lock rules. The proof of `release` has an interesting wrinkle: in order to give the resource R back to the lock, we must show that the lock is currently locked when `release` is called. We could accomplish this by adding a hold predicate in the style of Hobor et al. [2008], but we instead choose to take the same tack as is used by VST's built-in locks: we require that the lock invariant R be both *precise*, so that it can refer to at most one subheap of any heap, and *positive*, so that it must include ownership of at least one real resource in the heap. We can then prove the following lemma:

LEMMA 5.1. *If R is precise and positive, then $R * R \Rightarrow \bot$.*

Intuitively, each copy of R entails ownership of some specific resource, and a resource can only be owned once. From this lemma, it follows that if the releasing thread holds R (as required by the precondition of `release`), then the atomic location cannot also hold R, and so the lock cannot be in the unlocked state. This allows the lock invariant itself to do double duty as permission to release the lock.

In this way, we show that our spinlocks, implemented using either implementation of atomic exchange, have precisely the same specification as the built-in locks of VST. This is a good argument for the power of our atomic exchange specification, but in fact we can do much more with it, since spinlocks do not need to be history-aware. In the verification of the messaging system, presented in the next section, we demonstrate a more complex atomic exchange invariant that allows an atomic location to serve as a message-passing-style communication channel.

6 CORRECTNESS OF THE MESSAGING SYSTEM

We claim that our messaging system has three different kinds of useful properties.

(1) All operations are wait-free. This can be seen from the fact that the only non-thread-local operations are atomic exchanges, and these atomic operations are either outside of loops or in loops of fixed duration.
(2) Readers and writers that follow the protocol are guaranteed to get the most recent communication. This is a property we can prove with Concurrent Separation Logic, using ghost state (variables and histories) to relate the operations on the shared buffers with the values read and written by participants.
(3) Malicious readers or writers cannot compromise the correctness of the system. This requires a different kind of reasoning, more about security than correctness.

In the following sections, we will explain the proofs of the properties in the latter two categories.

6.1 Correctness for Valid Participants

Using VST, we can verify the correctness of the C code implementing the messaging protocol. "Correctness" here means memory safety, race freedom, and partial functional correctness (i.e., if a function terminates then it satisfies its specification). Aside from program state, the verification involves two aspects that do not appear in the code: the *permissions* on shared resources held by each thread, and the *ghost state* described above, which models the transfer of information between writer and readers via the location buffers. We give CSL specifications of each of the functions, and prove that the specifications hold on the function bodies.

6.1.1 Specification Elements. The implementation of the messaging system contains two sets of shared resources. The first consists of the location buffers `comm[N]`, one for each reader. Each buffer `comm[`r`]` is shared between reader r and the writer. The writer puts the ID of the most recently written data buffer into each location buffer, and reads either `Empty` (if the last message was successfully received) or the ID of the previously written data buffer (if the reader never received it). The reader puts `Empty` into its location buffer, and reads either the ID of the most recently written data buffer or, if none was written since the last read, its own `Empty` value. These operations are always performed via atomic exchange instructions, and no other operations are performed on the location buffers. If we are using the simulated atomic exchange of the previous section, then each buffer also has an associated lock.

The second set of shared resources consists of the data buffers `bufs[B]`. When a reader receives a data buffer ID b, it should be able to read from `bufs[`b`]` (but not from any other data buffer). When the writer knows that no reader is currently reading buffer b, it should be able to write to `bufs[`b`]`. In other words, the movement of buffer IDs through the messaging system also implies the movement of *permissions* to the corresponding data buffers. It is convenient to think of each data buffer as being divided into $N+1$ shares, with one earmarked for the writer and one for each reader[3]. Each share confers permission to read from the associated buffer, and only a thread in possession of all the shares can write to the buffer. When a reader receives buffer ID b, it also exchanges its share of the last-read buffer for a share of `bufs[`b`]`, allowing it to read. The writer always retains its own share of every data buffer, and when it publishes a buffer ID b, it distributes the remaining N shares into the corresponding location buffers (while retrieving any returned shares). When no readers know of buffer ID b, the writer holds all the remaining shares of `bufs[`b`]`, allowing it to write. Each of the N readers may hold a share of a different data buffer, and an additional buffer may have been distributed into the channels for reading, so the presence of $B = N + 2$ data buffers guarantees that there is always at least one buffer for which the writer holds all the shares.

[3]As is standard with CSL shares, we write $p \overset{\pi}{\longmapsto} v$ to assert that the current thread owns share π of p and p points to v; $p \overset{\pi_1}{\longmapsto} v * p \overset{\pi_2}{\longmapsto} v = p \overset{\pi_1 \cdot \pi_2}{\longmapsto} v$, where $\pi_1 \cdot \pi_2$ is the combination of the two shares.

To maintain the invariants and effect useful communication, the system also requires some ghost state, as described in Section 4. Each location buffer `comm[r]` has a history, with an associated label $g[r]$, which is shared between the writer, reader r, and the location buffer itself such that the location buffer has a complete history of all operations, and the reader and the writer each have a partial history. In fact, the reader should have the history of all reads from `comm[r]` (i.e., atomic exchanges in which the value written is `Empty`), and the writer should have the history of all writes (in which the value written is not `Empty`). To maintain this invariant, we use three ghost variables, each divided into two shares. Ghost variable $g_0[r]$ is shared between `comm[r]` and reader r, and maintains the invariant that the last value read by r is the same as the last value read from `comm[r]`. Ghost variables $g_1[r]$ and $g_2[r]$ are shared between `comm[r]` and the writer; $g_1[r]$ maintains the last value written to `comm[r]`, while $g_2[r]$ maintains the last value *known to have been taken* from `comm[r]`, i.e., the last value read before the last write. This last value matches the information stored in the `last_taken` array, and is used by the writer to track which buffers are held by readers: if the last buffer ID taken by reader r is b, then either r currently holds its share of `bufs[b]`, or the share has been returned to `comm[r]` but the writer has not yet collected it. The ghost variables capture the essence of the messaging protocol, by relating the local variables of the readers and writer to the complete history of operations performed on the location buffers.

The specifications of the reader and writer functions, then, will be written in terms of these elements: the location buffers, shares of the data buffers, partial histories of the location buffers, and the ghost variables that maintain the system-wide invariants. In particular, the atomic exchange invariant for each location buffer ties together the reference history, ghost variables, and communicated data buffer share:

$$R_r(h, v) \triangleq \exists b\ b_1\ b_2.\ \mathsf{last_two_reads}(h) = (b_1, b_2) \land g_0[r] \overset{.5}{\longmapsto} b_1 * g_1[r] \overset{.5}{\longmapsto} \mathsf{last_write}(h) * \\ g_2[r] \overset{.5}{\longmapsto} \mathsf{prev_taken}(h) * \text{if } v = \texttt{Empty} \text{ then } \texttt{bufs[}b_2\texttt{]} \overset{\pi_r}{\longmapsto} _ \text{ else } \texttt{bufs[}v\texttt{]} \overset{\pi_r}{\longmapsto} _$$

When the value v in the channel is a valid buffer ID, it holds a publication message from the writer, along with the rth share of data buffer v; when v is `Empty`, it holds an acknowledgment message from the reader, along with the rth share of the previously read buffer b_2.

6.1.2 *Reader Functions.*

Each reader `r` holds the following resources:

- full shares of its local variables `reading[r]` and `last_read[r]`, the latter of which holds the ID of its most recently read data buffer b_0
- a half share of the atomic location buffer `comm[r]`, with the partial history h of reads performed on it
- the `r`th share π_r of the buffer b_0
- a half share of the ghost variable g_0`[r]`, which relates b_0 to the complete history of `comm[r]`

The `start_read` function exchanges `r`'s share of b_0 for a share of the most recently written data buffer. If no buffer has been written since the last `start_read`, it retains its share of b_0 instead. In the process, it updates its history and ghost variable to reflect the new read.

The main work of `start_read` is done in the atomic exchange. Recall from Section 5 that the specification of atomic exchange is parameterized by a pre- and postcondition. The precondition provided by `start_read` contains the ghost variable g_0`[r]` and the share of the buffer b_0; the postcondition contains the new value of the ghost variable and the share of the new buffer, along with a guarantee that the new value is the most recent read in the extended history. Because the value written is always `Empty`, this leads to two possibilities: either the value read is a non-`Empty` value b, g_0`[r]` is now b, and the received buffer is b; or the value read is `Empty`, g_0`[r]` is still b_0, and the received buffer is b_0. The remainder of `start_read` distinguishes between these two cases and

$$\left\{\begin{array}{l}\texttt{reading[r]} \mapsto _ * \texttt{last_read[r]} \mapsto b_0 * \texttt{comm[r]} \odot\!\xrightarrow{.5} (g[\texttt{r}], 0, R_{\texttt{r}}, h) * \\ \texttt{bufs}[b_0] \xmapsto{\pi_{\texttt{r}}} _ * g_0[\texttt{r}] \xmapsto{.5} b_0 \wedge \mathsf{latest_read}(h, b_0)\end{array}\right\}$$

```
buf_id start_read(int r) {
  buf_id b = atomic_exchange (&comm[r], Empty);
```

$$\left\{\begin{array}{l}b = v \wedge b' = (\text{if } v = \texttt{Empty} \text{ then } b_0 \text{ else } v) \wedge \texttt{reading[r]} \mapsto _ * \texttt{last_read[r]} \mapsto b_0 * \\ \exists \text{fresh } t'.\ \texttt{comm[r]} \odot\!\xrightarrow{.5} (g[\texttt{r}], 0, R_r, h \cup \{(t', [v \rightarrow \texttt{Empty}])\}) * \texttt{bufs}[b'] \xmapsto{\pi_{\texttt{r}}} _ * g_0[\texttt{r}] \xmapsto{.5} b' \wedge \\ \mathsf{latest_read}(h \cup \{(t', [v \rightarrow \texttt{Empty}])\}, b')\end{array}\right\}$$

```
  if (b>=0 && b<B)
     last_read[r] = b;
  else
     b = last_read[r];
  reading[r] = b;
  return b;
}
```

$$\left\{\begin{array}{l}\lambda b.\ \texttt{reading[r]} \mapsto b * \texttt{last_read[r]} \mapsto b * \\ \exists \text{fresh } t'.\ \texttt{comm[r]} \odot\!\xrightarrow{.5} (g[\texttt{r}], 0, R_r, h \cup \{(t', [v \rightarrow \texttt{Empty}])\}) * \texttt{bufs}[b] \xmapsto{\pi_{\texttt{r}}} _ * \\ g_0[\texttt{r}] \xmapsto{.5} b \wedge b = (\text{if } v = \texttt{Empty} \text{ then } b_0 \text{ else } v) \wedge \mathsf{latest_read}(h \cup \{(t', [v \rightarrow \texttt{Empty}])\}, b)\end{array}\right\}$$

Fig. 6. The `start_read` function annotated with separation logic specification

updates `last_read` and `reading` as appropriate. By the end of the function, the reader holds the same resources as before, with b_0 replaced by b if the read was successful.

$\{\texttt{reading[r]} \mapsto _\}$

```
void finish_read(int r) {
  reading[r] = Empty;
}
```

$\{\texttt{reading[r]} \mapsto \texttt{Empty}\}$

Fig. 7. The `finish_read` function annotated with separation logic specification

The `finish_read` function does not involve any shared resources, and its specification and verification are straightforward.

6.1.3 The `start_write` *Function.* The `start_write` function also does not involve any shared resources, but its behavior is slightly more complicated than `finish_read`: it searches for a buffer ID b that is not equal to `last_given` and does not appear anywhere in `last_taken`, and sets `writing` equal to b. An available buffer is guaranteed to exist by the pigeonhole principle: the number of used data buffers is at most $N + 1$, since the length of `last_taken` is N, and the total number of data buffers is $B = N + 2$.

The top-level correctness property of `start_write` is that the data buffer `bufs[`b`]` is not in use by any reader, but its specification only guarantees that b is not in `last_taken` or equal to `last_given`. In order to conclude that the writer can actually write to `bufs[`b`]`, we need to know

$\{\texttt{writing} \mapsto _ * \texttt{last_given} \mapsto b_0 * \texttt{last_taken} \mapsto \mathit{lasts}\}$

```
buf_id start_write(){
  int available[B];
  for (i=0; i<B; i++)
       available[i] = 1;
  available[last_given]=0;
  for (r=0; r<N; r++) {
    if (last_taken[r]!=Empty)
      available[last_taken[r]]=0;
  }
  for (i=0; i<B; i++)
      if (available[i]) {
        writing=i; return i;
      }
  assert(0);
}
```

$\{\lambda b.\, \texttt{writing} \mapsto b * \texttt{last_given} \mapsto b_0 * \texttt{last_taken} \mapsto \mathit{lasts} \wedge b \notin \{b_0\} \cup \mathit{lasts}\}$

Fig. 8. The `start_write` function annotated with separation logic specification

something about the permissions held by the writer. At each point in the execution of the writer, we can calculate from its local variables the share it owns of each data buffer b, as follows:

- If b is `last_given`, then the writer has only the writer share of `bufs`[b].
- Otherwise, the writer has all shares of `bufs`[b] except for the reader shares of readers r for which `last_taken`[r]= b.

We will write the share thus computed as $\pi_b(b_0, \mathit{lasts})$, where b_0 is the value of `last_given` and *lasts* is the list of values in `last_taken`.

If a buffer b is not equal to `last_given` and does not appear in `last_taken`, then π_b is the full share. Then we can be sure that the writer has all shares of `bufs`[b], and can write to it. This is sufficient to guarantee that `bufs`[b] is not in use by any reader, even without global knowledge of the buffers currently being read: if any reader was reading from `bufs`[b], then it would have some share of `bufs`[b], which is ruled out by the invariant that π_b accurately computes the writer's share.

6.1.4 The `finish_write` *Function.* When `finish_write` is called, the writer holds the following resources:

- full shares of its local variables: `last_given`, which holds the ID b_0 of the previously written data buffer; the array `last_taken`; and `writing`, which holds a value b that is not equal to b_0 and appears nowhere in `last_taken` (we also maintain the invariant that b_0 appears nowhere in `last_taken`)
- a share of each atomic location buffer `comm[r]`, with the partial history h[`r`] of writes performed on it (the $\circledast$ operator represents iterated separating conjunction)
- for each reader `r`, half shares of the ghost variables g_1[`r`] and g_2[`r`], which relate b_0 and `last_taken[r]` respectively to the complete history of `comm[r]`

$$\left\{\begin{array}{l}\texttt{writing} \mapsto b * \texttt{last_given} \mapsto b_0 * \texttt{last_taken} \mapsto \mathit{lasts} * \\ \circledast_{\texttt{r}} \left(\texttt{comm[r]} \odot\!\!\xrightarrow{.5} (g[\texttt{r}], 0, R_{\texttt{r}}, h[\texttt{r}]) * g_1[\texttt{r}] \overset{.5}{\longmapsto} b_0 * g_2[\texttt{r}] \overset{.5}{\longmapsto} \mathit{lasts}[\texttt{r}]\right) * \\ \circledast_{b'}\texttt{bufs}[b'] \xmapsto{\pi_b(b_0, \mathit{lasts})} _ \wedge b_0 \notin \mathit{lasts} \wedge b \notin \{b_0\} \cup \mathit{lasts}\end{array}\right\}$$

```
void finish_write(){
  for (r=0; r<N; r++) {
    buf_id b = atomic_exchange (&comm[r], writing);
    if (b==Empty)
       last_taken[r] = last_given;
  }
  last_given = writing;
  writing = Empty;
}
```

$$\left\{\begin{array}{l}\texttt{writing} \mapsto \texttt{Empty} * \texttt{last_given} \mapsto b * \exists \mathit{lasts}'.\ \texttt{last_taken} \mapsto \mathit{lasts}' * \\ \circledast_{\texttt{r}} \left(\begin{array}{l}\exists \text{fresh } t'.\ \exists v'.\ \texttt{comm[r]} \odot\!\!\xrightarrow{.5} (g[\texttt{r}], 0, R_{\texttt{r}}, h[\texttt{r}] \cup \{(t', [v' \to b])\}) * \\ g_1[\texttt{r}] \overset{.5}{\longmapsto} b * g_2[\texttt{r}] \overset{.5}{\longmapsto} \mathit{lasts}'[\texttt{r}]\end{array}\right) * \\ \circledast_{b'}\texttt{bufs}[b'] \xmapsto{\pi_{b'}(b, \mathit{lasts}')} _ \wedge b \notin \mathit{lasts}'\end{array}\right\}$$

Fig. 9. The `finish_write` function annotated with separation logic specification

- an appropriate share π_b of each buffer b, determined by `last_given` and `last_taken` as described above

Once again, all the hard work is done in the series of atomic exchanges on each location buffer. The writer publishes the ID b of the newly written data buffer to each channel `comm[r]`, and collects a share of any other data buffer that may be in `comm[r]`. There are two cases in which `comm[r]` may already contain a share: reader `r` may not have received b_0 at all yet, or `r` may have received b_0 and returned its share of another buffer in exchange. In each of these cases, retrieving the share of the buffer in the channel maintains the invariant on the shares held by the writer. Proving this is the tricky part, accounting for 470 lines of Coq code out of a total of about 2000 lines for the core functions; a sketch of the proof can be found in Section A.2.

6.1.5 Verifying a Client. The specifications for the messaging functions are complex enough that it is not obvious that they allow the system to be used as desired. If the preconditions are unsatisfiable, then the specifications may be correct but not useful. To ensure that the messaging system as verified is usefully correct, we have also verified a simple client. The client consists of readers and a writer that use the functions straightforwardly: the writer repeatedly calls `start_write`, writes a value to the indicated data buffer, and calls `finish_write`, while the reader repeatedly calls `start_read`, reads from the indicated data buffer, and calls `finish_read`. In order for these functions to be verifiable, it must be the case that 1) the reader and writer invariants are preserved by normal use of the functions, and 2) the permissions obtained by calling the functions are sufficient to allow the readers to read and the writer to write the chosen data buffers.

The code and proof outlines for the client are shown in Section A.3. Each of these properties follows quickly from the specifications proved above. The `start_read` and `finish_write` specifications have postconditions of the same shape as their preconditions, modulo changes in

the currently and most recently accessed buffers. For `start_read`, no further work is needed to reestablish the precondition, and `finish_read` is effectively a noop; for `finish_write`, a following call to `start_write` is needed to find a new writeable buffer and reestablish the precondition. When `start_read(r)` returns a buffer ID `b`, it also provides the share π_r of `bufs[b]`, which allows reader `r` to read from the buffer; when `start_write()` returns a buffer ID `b`, the postcondition in combination with the definition of π_b guarantees that the writer holds all shares of `bufs[b]` and can write to it.

Topic	loc
PCMs and ghost state	900
`atomic_exchange`	250
locks implemented with `atomic_exchange`	230
messaging function specifications	330
supporting functions	220
`start_read` proofs	220
`finish_read` proofs	10
`start_write` proofs	180
`finish_write` proofs	830
client	400
total	3600

Table 1. Size of Coq definitions and proofs, by topic.

6.1.6 Coq Development. All the proofs described above were formalized in Coq, using VST's facilities for proving separation logic specifications of C functions. VST provides automation for simple separation logic reasoning, so that a single `forward` command can be used to process simple commands (assignments, loads and stores, etc.). Most of the proof effort went into showing that preconditions of functions (including `atomic_exchange`) were satisfied and that lock and loop invariants were maintained. The largest single section of the proof is the lemma that the loop invariant for `finish_write` is preserved when the writer sends a buffer ID to a reader and receives its response; this proof involves updating the permissions held by the writer with the messages received from each reader in sequence. The proof of the lemma is about 470 lines, and the remaining reasoning about the `finish_write` function is another 360 lines.

6.2 Noninterference Properties

We can reason informally about the correctness of the system even in the presence of unverified readers and/or writers, which may execute arbitrary code instead of or in addition to the four core functions. As described in Section 2, we can use a pattern of virtual memory protections to isolate readers from other readers and writers from other writers. The goal is that a malicious reader cannot prevent a writer from writing, a malicious reader cannot interfere with other readers, and a malicious writer cannot stop readers from reading from other writers.

For readers, the only shared data that reader `r` has write access to is `comm[r]`. It can write a value other than `Empty` to `comm[r]`, but any non-`Empty` value is treated by the writer as a sign that the sent message has not yet been received (which is valid behavior for a correct reader as well). Even if the reader (valid or otherwise) never acknowledges a new message and holds onto its current data buffer forever, the messaging system can continue with one fewer data buffer and one fewer

reader. A malicious reader can also perform a blocking synchronization operation on `comm[r]`, but this will not cause the `atomic_exchange` operations of valid participants to block.

A malicious writer can write to any of its associated buffers. This allows it to store bad data in data buffers, and communicate that data to readers; it also allows it to send bad locations, i.e., out-of-range buffer IDs. However, the bounds checking in `start_read` means that out-of-range buffer IDs are treated as `Empty`, preventing out-of-bounds access to the data buffer array. A writer could also perform blocking synchronization operations on either kind of buffer, but it remains the case that all operations performed by valid readers are nonblocking. If a reader is part of multiple separate messaging systems with different writers, then even if one of the writers is malicious, the reader cannot be forced to block or prevented from reading good data from the other writers. So the worst a malicious writer can do is to communicate incorrect data; it cannot interfere with the ability of valid readers to read correct data from other valid writers.

7 PRACTICAL USE AND PERFORMANCE

Michael Whalen (University of Minnesota), Pape Sylla (HRL Laboratories, LLC) and Aleksey Nogin (HRL) have incorporated support for the messaging system into the Trusted Build (TB) glue code synthesis system [Cofer and Whalen 2013], allowing automated synthesis of both the messaging code and the requisite virtual memory sharing and protections. The current implementation supports communication between software components running under the seL4 operating system [Klein et al. 2009], between processes running under Linux, as well as seL4 components communicating with Linux processes within a Linux virtual machine running under seL4. The HRL team has developed prototype high-assurance waypoint navigation, obstacle avoidance, and roll-over prevention code for a Heavy Equipment Transport (HET) truck that consisted of several seL4 components with the large majority of intercomponent connections utilizing the messaging protocol. The HRL team has also developed high-assurance software for a small ground robot, using the messaging protocol to safely connect the untrusted Linux processes inside a VM with trusted seL4 components. As of the time of writing, the HET truck has been evaluated by an independent Red Team, with no major vulnerabilities found; the security evaluation of the ground robot is still in progress. In our experience, the messaging protocol was a very convenient means for connecting independently scheduled and potentially mutually distrusting software components in a real-time system. One change was added for practical convenience—in the TB implementation, the `start_write` function does a memcpy from the last written buffer to the new buffer. This allows the writer code to create the new state by modifying the old one, rather than having to create it from scratch.

To get a better idea of the performance overhead of the messaging protocol, we developed the following experiment: 20 Linux processes use 20 instances of the messaging system to communicate among themselves. Each process is a writer for 1 system, and a reader for the remaining 19. In a loop, each process 1) executes `start_read`/`start_write` for all instances, 2) computes $m =$ `min`(contents of all its 20 input and output buffers), 3) writes $m+1$ into its output buffer, 4) executes `finish_read`/`finish_write` for all instances. Effectively, each process keeps sending the message 1 until it sees that all its peers have done the same, then it keeps sending 2 until it sees that all its peers have done the same, and so on. We executed this code on a 2x6-core (24 threads) dual 2GHz Intel Xeon E5-2620 CPU system. In a 10 minute run, we observed an average loop cycle time of approximately 4.37 μs, with message values getting incremented approximately every 1.83 cycles (8.02 μs) on average. As a baseline comparison, we ran the exact same code with the messaging synchronizations removed—simply using the first buffer in place of `start_` function calls and not calling the `finish_` functions. This time we observed a cycle time of approximately 1.07 μs, with message values getting incremented approximately every 1.85 cycles (1.97 μs). Each cycle performs

38 synchronization operations (19 in `finish_write` and 1 in each of the 19 `start_read` calls), so each synchronization operation was adding on average approximately 0.087 μs of overhead.

8 RELATED WORK

Parkinson et al. [2007] were the first to verify a program with an atomic operation (compare-and-swap), using an ad-hoc extension of the atomic rule from base CSL. The CAS rule of Relaxed Separation Logic [Vafeiadis and Narayan 2013] bears more of a resemblance to our rule for atomic exchange, although it is more complicated to allow for non-well-synchronized use; our rule is meant to be applied only to atomics that guarantee a total order on operations on the location. RSL is also mechanized in Coq, as are many other comparable concurrent separation logics, including Iris, FCSL, GPS, and FSL++ (discussed later in this section). However, none of these logics are connected to a full-scale formalization of a real-world language, including a verified compiler to assembly, as VST is, and so correctness proofs of algorithms are not directly related to real executing programs.

Many recent concurrent separation logics have taken various approaches to atomic operations and ghost state; our treatment of ghost state is based on that of Iris [Jung et al. 2016]. The main difference between our CSL and Iris (aside from C-specific features) is the treatment of invariants. Invariants in Iris are first-class, named, persistent resources: Hoare triples are proved in the context of an environment of invariants, and there are special rules for opening invariants to use their contents and then closing them once they are reestablished. In our logic, invariants only exist in the form of lock and atomic-location invariants, which are attached to specific heap locations, must be divided into shares and passed to any threads that use them, and are created and destroyed by the commands that create and destroy the locks/atomic locations. This allows us to consistently treat all assertions as either pure logical state or separation-style resources, and fits naturally with the use of synchronization primitives in C. In Iris, invariants may be conditioned on the initialization of heap resources but need not be associated with them; permission-passing reasoning has been shown to be derivable from this kind of invariant, and our lock and atomic invariants could be derived in Iris as well.

Our approach to histories as ghost state is influenced by both VerCors [Blom et al. 2015], which has shared histories as first-class objects and has been used to verify concurrent counter and (coarse-grained) set implementations, and FCSL [Sergey et al. 2015], which has been used to verify fine-grained concurrent data structures and work-sharing algorithms. Our partial histories are sets of timestamped operations, as in FCSL, while our reference histories are lists of operations, as in VerCors; the combination of the two serves as an alternative to the *subjectivity* of FCSL, which requires threads to track both “self” and “other” histories.

Tassarotti et al. [2015] verified the Read-Copy-Update (RCU) protocol using the concurrent separation logic GPS. They use acquire and release atomics to establish a linked list with a single writer and multiple readers; the verification is similarly based on passing shares between readers and the writer. Aside from the differences between the two protocols, there are differences between the logics that cause the proofs to look significantly different. The GPS verification uses a special category of protocol assertions to describe the effects of atomic operations, and several types of ghost state, including exchanges and permission tokens, are used to describe the transfer of resources between threads. In our setting, histories and ghost variables allow us to perform the same kind of reasoning, and it may be possible to implement protocol assertions on top of histories. GPS also explicitly accounts for relaxed memory effects, while we require that programs be sufficiently well synchronized to avoid relaxed behaviors.

Another logic for low-level atomics is FSL++ [Doko and Vafeiadis 2017], a descendant of RSL that includes support for relaxed atomics in combination with fences. Extra modalities allow programs

to promise to give up resources or expect to gain them, with these promises and expectations only realized after the appropriate fences. FSL++ has been used to verify an atomic reference counter, which involves permission transfer and ghost state, and is one of the most complicated fine-grained concurrent algorithms to be verified with concurrent separation logic, particularly in terms of relaxed memory effects.

Fine-grained concurrent programs have also been analyzed with automatic verification tools. McKenney [2007] used Promela and Spin [Holzmann 1990] to verify QRCU, a version of RCU with higher read overhead but faster writing. Specifically, he built a Promela model of RCU and verified that writers correctly determine when all readers are ready to progress, for several fixed small numbers of readers and writers. Additional informal reasoning is needed to show that correctness generalizes to larger numbers of readers, and the details of real C implementations are not taken into account. VeriFast, an automatic separation-logic-based program verifier for C and Java, has also been used to verify fine-grained concurrent programs [Jacobs and Piessens 2011]. The CSL used is a predecessor of Iris, and is formalized on Coq on a simple concurrent language; its application to C code, on the other hand, has not been formally verified.

9 CONCLUSIONS AND FUTURE WORK

We have described the implementation of a messaging system that allows writers to send messages to readers concurrently, without any party waiting for any other. We have formally proven that the system is race-free, memory-safe, and functionally correct when all participants follow the messaging protocol. To the best of our knowledge, this is the first verified implementation of a shared-memory communication protocol at the C level (and in fact, VST guarantees the correctness of the protocol even when compiled to assembly). By adding virtual memory protections at initialization, we can limit the potential harm caused by incorrect or malicious components: readers cannot interfere with any other component, and writers cannot force readers to block or interfere with communication between readers and other writers. The formal proof of correctness relies on a novel and general axiomatization of atomic exchange, which we have shown is consistent with the intuitive specification of atomic exchange in terms of a critical-section read and write, and can be used to implement spinlocks with arbitrary CSL resource invariants.

While our specific use case is communication between sensors and control systems of an autonomous vehicle, our messaging protocol could also be built into an operating system as a secure form of inter-process communication, or used in any application requiring high-assurance communication between components in the presence of untrusted components. For instance, in distributed systems such as the Robot Operating System [Quigley et al. 2009], when processes are on the same machine, they can shortcut the network and communicate via shared memory; the messaging system would provide a secure and reliable mechanism for this kind of communication in multiple-reader settings (such as publish/subscribe communication).

Our techniques for verifying nonblocking concurrent programs should be easily extensible to other atomic operations, including load, store, and compare-and-swap, and perhaps to synchronization modes weaker than SC, as long as the synchronization patterns can be analyzed in terms of transfer of permissions. Our immediate next goal is to use these extensions to verify fine-grained concurrent data structures and algorithms written in C. Another interesting problem is to formalize the informal reasoning about the effects of malicious components in Section 6.2, and connect it to the existing Coq proofs. We could, for instance, use ghost-state tokens to track which readers and writers are trusted, and then assume that a location buffer follows the protocol only if both parties to it are trusted. We could then verify clients that read both trusted and untrusted data, where untrusted data cannot be assumed to be correct and may be changed without warning.

ACKNOWLEDGMENTS

This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0293, by the National Science Foundation under grant CCF-152160, by a grant from Intel Corporation, and by the United States Air Force and DARPA under contract number FA8750-12-C-0281. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, NSF, Department of Defense, or the U.S. Government.

REFERENCES

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers.* Cambridge University Press, New York, NY, USA.

Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. 2015. History-Based Verification of Functional Behaviour of Concurrent Programs. In *Proceedings of the 13th International Conference on Software Engineering and Formal Methods (SEFM 2015) (Lecture Notes in Computer Science)*, Radu Calinescu and Bernhard Rumpe (Eds.), Vol. 9276. Springer, 84–98. https://doi.org/10.1007/978-3-319-22969-0_6

Darren Cofer and Michael Whalen. 2013. Secure Mathematically-Assured Composition of Control Models (SMACCM). (2013). https://wiki.sei.cmu.edu/aadl/images/f/f6/SMACCM-TA4-whalen-42013.pdf

P. J. Courtois, F. Heymans, and D. L. Parnas. 1971. Concurrent Control with "Readers" and "Writers". *Commun. ACM* 14, 10 (Oct. 1971), 667–668. https://doi.org/10.1145/362759.362813

Marko Doko and Viktor Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *Proceedings of the 26th European Symposium on Programming (ESOP 2017) (Lecture Notes in Computer Science)*, Hongseok Yang (Ed.), Vol. 10201. 448–475. https://doi.org/10.1007/978-3-662-54434-1_17

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *Proceedings of the 17th European Symposium on Programming (ESOP 2008) (Lecture Notes in Computer Science)*, Sophia Drossopoulou (Ed.), Vol. 4960. Springer, 353–367. https://doi.org/10.1007/978-3-540-78739-6_27

Gerard J. Holzmann. 1990. Design and Validation of Protocols. *Tutorial Computer Networks and ISDN Systems* 25 (1990), 981–1017.

Bart Jacobs and Frank Piessens. 2011. Expressive Modular Fine-grained Concurrency Specification. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11).* ACM, New York, NY, USA, 271–282. https://doi.org/10.1145/1926385.1926417

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order Ghost State. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016).* ACM, New York, NY, USA, 256–269. https://doi.org/10.1145/2951913.2951943

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09).* ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf

Paul E. McKenney. 2007. Using Promela and Spin to verify parallel algorithms. (1 August 2007). Available: http://lwn.net/Articles/243851/.

Paul E. McKenney and John D. Slingwine. 1998. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems.* Las Vegas, NV, 509–518.

Peter W. OHearn. 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

Susan Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs I. *Acta Informatica* 6, 4 (1976), 319–340. https://doi.org/10.1007/BF00268134

Matthew Parkinson, Richard Bornat, and Peter O'Hearn. 2007. Modular Verification of a Non-blocking Stack. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07).* ACM, New York, NY, USA, 297–302. https://doi.org/10.1145/1190216.1190261

Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software.*

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *Proceedings of the 24th European Symposium on Programming (ESOP 2015) (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer, 333–358. https://doi.org/10.1007/978-3-662-46669-8_14

Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. 2015. Verifying Read-copy-update in a Logic for Weak Memory. (2015), 110–120. https://doi.org/10.1145/2737924.2737992

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: a program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2013)*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 867–884. https://doi.org/10.1145/2509136.2509532

A APPENDIX

A.1 Atomic Exchange (Section 5)

Theorem A.1 (Atomic Exchange). *Let* `simulate_atomic_exchange` *be defined as in Section 5.1. Then the following rule holds:*

$$\frac{\mathsf{AEX_triple}(P, Q, R)}{\begin{array}{l}\{\ell \boxdot\!\!\rightarrow \mathsf{AEX_inv}(x, g, i, R) * \mathsf{hist}(h)@g * P(h, v)\}\ \mathtt{simulate_atomic_exchange}(x, \ell, v)\\ \{\lambda v'.\ \ell \boxdot\!\!\rightarrow \mathsf{AEX_inv}(x, g, i, R) * \exists t'.\ \mathsf{hist}(h \cup \{(t', [v' \to v])\})@g *\\ \ Q(h \cup \{(t', [v' \to v])\}, v')\}\end{array}}$$

Proof. We can prove the following sequence of CSL triples:

$\{\ell \boxdot\!\!\rightarrow \mathsf{AEX_inv}(x, g, i, R) * \mathsf{hist}(h)@g * P(h, v)\}$

```
simulate_atomic_exchange(int *x, lock l, int v){
    acquire(l);
```

$$\left\{\begin{array}{l}\exists r\ v'.\ \mathsf{apply}(i, r) = v' \land x \mapsto v' * \mathsf{reference}(r)@g * R(r, v') * \ell \boxdot\!\!\rightarrow \mathsf{AEX_inv}(x, g, i, R) *\\ \mathsf{hist}(h)@g * P(h, v)\end{array}\right\}$$

```
    w = *x;
```

$$\left\{\begin{array}{l}w = v' \land \mathsf{apply}(i, r) = v' \land x \mapsto v' * \mathsf{reference}(r)@g * R(r, v') * \ell \boxdot\!\!\rightarrow \mathsf{AEX_inv}(x, g, i, R) *\\ \mathsf{hist}(h)@g * P(h, v)\end{array}\right\}$$

```
    *x = v;
```

$$\left\{\begin{array}{l}w = v' \land \mathsf{apply}(i, r) = v' \land x \mapsto v * \mathsf{reference}(r)@g * R(r, v') * \ell \boxdot\!\!\rightarrow \mathsf{AEX_inv}(x, g, i, R) *\\ \mathsf{hist}(h)@g * P(h, v)\end{array}\right\}$$

$$\left\{\begin{array}{l}w = v' \land \mathsf{apply}(i, r + [v' \to v]) = v \land x \mapsto v * \mathsf{reference}(r + [v \to v'])@g *\\ R(r + [v' \to v], v) * \ell \boxdot\!\!\rightarrow \mathsf{AEX_inv}(x, g, i, R) * \mathsf{hist}(h \cup \{(t', [v' \to v])\})@g *\\ Q(h \cup \{(t', [v' \to v])\}, v')\end{array}\right\}$$

```
    release(l);
```

$\{w = v' \land \ell \boxdot\!\!\rightarrow \mathsf{AEX_inv}(x, g, i, R) * \mathsf{hist}(h \cup \{(t', [v' \to v])\})@g * Q(h \cup \{(t', [v' \to v])\}, v')\}$

```
    return w;
}
```

$$\left\{\begin{array}{l}\lambda v'.\ \ell \boxdot\!\!\rightarrow \mathsf{AEX_inv}(x, g, i, R) * \exists \mathsf{fresh}\ t'.\ \mathsf{hist}(h \cup \{(t', [v' \to v])\})@g *\\ Q(h \cup \{(t', [v' \to v])\}, v')\end{array}\right\}$$

Before the call to `release`, we use the definition of AEX_triple along with the view shift rule to swap P for Q while adding an entry to both histories. The new timestamp t' is the newest index in the reference history, but we transform it into an existentially quantified variable at the end of the function, forgetting all information about the reference history. □

Lemma 1. *If R is precise and positive, then $R * R \Rightarrow \bot$.*

Proof. Suppose we have a heap H such that $R*R$ holds on H. Then there are two non-overlapping subheaps of H, H_1 and H_2, such that R holds on both. Because R is precise, there is exactly one subheap H' of H on which it can hold, so $H_1 = H_2 = H'$. Thus H' does not overlap with itself, i.e., it is an empty heap. But because R is positive and holds on H', there must be some real resource in H' described by R, and so H' cannot be empty. □

Theorem A.2 (Lock Acquire). *Let* `acquire` *be defined as in Section 5.3. Then*

$$\{\mathsf{lock}(\ell, g, R)\}\ \mathtt{acquire}(\ell)\ \{\mathsf{lock}(\ell, g, R) * R\}$$

Proof. We need to find a loop invariant I such that $\mathsf{lock}(\ell, g, R) \Rightarrow I$ and $I \Rightarrow \mathsf{lock}(\ell, R) * R$. The natural choice is $I = \mathsf{lock}(\ell, g, R) * \text{if } r = 0 \text{ then } R \text{ else emp}$. To show that I is preserved by the `atomic_exchange`, we must present a pre- and postcondition that form an AEX_triple with $R_{\mathsf{lock}}(R) \triangleq (\text{if } v = 0 \text{ then } R \text{ else emp})$. Before the exchange, r is always non-zero, so $\lambda(h, v).\ v \neq 0 \wedge \mathsf{emp}$ is a suitable precondition. Then we must have that $v \neq 0 \wedge \mathsf{emp} * (\text{if } v' = 0 \text{ then } R \text{ else emp}) \Rrightarrow Q(h', v') * \mathsf{emp}$, leading to a valid triple when $Q(h, v) = (\text{if } v = 0 \text{ then } R \text{ else emp})$, which after the assignment of the return value to r immediately reestablishes I. □

Theorem A.3 (Lock Release). *Let* `release` *be defined as in Section 5.3. Then*

$$\{\mathsf{lock}(\ell, g, R) * R\}\ \ \mathtt{release}(\ell)\ \{\mathsf{lock}(\ell, g, R)\}$$

Proof. The entire function body is an `atomic_exchange`, so we need only find a suitable AEX_triple. Before the exchange, we know that R holds and we will write 0, so we choose the precondition $\lambda(h, v).\ v = 0 \wedge R$. Then we must have that $v = 0 \wedge R * (\text{if } v' = 0 \text{ then } R \text{ else emp}) \Rrightarrow Q(h', v') * R$. Lemma 5.1 allows us to rule out the case in which $v' = 0$, so the implication holds when $Q(h, v) = (v \neq 0 \wedge \mathsf{emp})$. Thus, after the `atomic_exchange`, the only resource left is $\mathsf{lock}(\ell, R)$, as desired. □

A.2 Correctness of the Messaging System (Section 6.1)

For each of the four major functions of the messaging system, we present its annotation with CSL assertions, and explain any steps that consist of more than just application of the CSL rule for the associated command.

Theorem A.4. *The* `start_read` *function meets its specification.*

Proof.

$$\left\{\begin{array}{l}\mathtt{reading[r]} \mapsto _ * \mathtt{last_read[r]} \mapsto b_0 * \mathtt{comm[r]} \odot\!\!\xrightarrow{.5} (g[\mathsf{r}], 0, R_{\mathsf{r}}, h) * \\ \mathtt{bufs}[b_0] \overset{\pi_{\mathsf{r}}}{\longmapsto} _ * g_0[\mathsf{r}] \overset{.5}{\longmapsto} b_0 \wedge \mathsf{latest_read}(h, b_0)\end{array}\right\}$$

```
buf_id start_read(int r) {
  buf_id b = atomic_exchange (&comm[r], Empty);
```

$$\left\{\begin{array}{l}b = v \wedge b' = (\text{if } v = \mathtt{Empty} \text{ then } b_0 \text{ else } v) \wedge \mathtt{reading[r]} \mapsto _ * \mathtt{last_read[r]} \mapsto b_0 * \\ \exists \text{fresh } t'.\ \mathtt{comm[r]} \odot\!\!\xrightarrow{.5} (g[\mathsf{r}], 0, R_{\mathsf{r}}, h \cup \{(t', [v \rightarrow \mathtt{Empty}])\}) * \mathtt{bufs}[b'] \overset{\pi_{\mathsf{r}}}{\longmapsto} _ * g_0[\mathsf{r}] \overset{.5}{\longmapsto} b' \wedge \\ \mathsf{latest_read}(h \cup \{(t', [v \rightarrow \mathtt{Empty}])\}, b')\end{array}\right\}$$

```
  if (b>=0 && b<B)
     last_read[r] = b;
  else
     b = last_read[r];
```

$$\left\{\begin{array}{l} b = b' \land b' = (\text{if } v = \texttt{Empty} \text{ then } b_0 \text{ else } v) \land \texttt{reading[r]} \mapsto _ * \texttt{last_read[r]} \mapsto b' * \\ \texttt{comm[r]} \odot\!\!\xrightarrow{.5} (g[\texttt{r}], 0, R_\texttt{r}, h \cup \{(t', [v \rightarrow \texttt{Empty}])\})) * \texttt{bufs[}b'\texttt{]} \xmapsto{\pi_\texttt{r}} _ * g_0[\texttt{r}] \xmapsto{.5} b' \land \\ \text{latest_read}(h \cup \{(t', [v \rightarrow \texttt{Empty}])\}, b') \end{array}\right\}$$

```
  reading[r] = b;
  return b;
}
```

$$\left\{\begin{array}{l} \lambda b.\ \texttt{reading[r]} \mapsto b * \texttt{last_read[r]} \mapsto b * \\ \exists \text{fresh } t'.\ \texttt{comm[r]} \odot\!\!\xrightarrow{.5} (g[\texttt{r}], 0, R_r, h \cup \{(t', [v \rightarrow \texttt{Empty}])\}) * \texttt{bufs[}b\texttt{]} \xmapsto{\pi_\texttt{r}} _ * \\ g_0[r] \xmapsto{.5} b \land b = (\text{if } v = \texttt{Empty} \text{ then } b_0 \text{ else } v) \land \text{latest_read}(h \cup \{(t', [v \rightarrow \texttt{Empty}])\}, b) \end{array}\right\}$$

The change in resources occurs entirely via the `atomic_exchange`; the remaining lines serve to update the reader's internal accounting. The precondition for the `atomic_exchange` is $\lambda(h, v).\ v = \texttt{Empty} \land \text{latest_read}(h, b_0) \land \texttt{bufs[}b_0\texttt{]} \xmapsto{\pi_\texttt{r}} _ * g_0[\texttt{r}] \xmapsto{.5} b_0$. The atomic exchange invariant R_r presents two cases: either the value in `comm[r]` is `Empty` and `comm[r]` contains π_r of the second most recently read buffer, or the value is some non-`Empty` v and `comm[r]` contains π_r of buffer v. In the former case, the reader retains its share of buffer b_0; in the latter, it exchanges it for the share of buffer v. This leads to the two cases in the postcondition of the `atomic_exchange`. □

Theorem A.5. *The* `finish_read` *function meets its specification.*

Proof.

$\{\texttt{reading[r]} \mapsto _\}$

```
void finish_read(int r) {
  reading[r] = Empty;
}
```

$\{\texttt{reading[r]} \mapsto \texttt{Empty}\}$

The proof is simply the application of the store rule. □

Theorem A.6. *The* `start_write` *function meets its specification.*

Proof.

$$\{\texttt{writing} \mapsto _ * \texttt{last_given} \mapsto b_0 * \texttt{last_taken} \mapsto \mathit{lasts}\}$$

```
buf_id start_write(){
  int available[B];
  for (i=0; i<B; i++)
      available[i] = 1;
  available[last_given]=0;
  for (r=0; r<N; r++) {
    if (last_taken[r]!=Empty)
      available[last_taken[r]]=0;
  }
```

$$\left\{\begin{array}{l} \texttt{writing} \mapsto _ * \texttt{last_given} \mapsto b_0 * \texttt{last_taken} \mapsto \mathit{lasts} \land \\ \forall i.\ \texttt{available[}i\texttt{]} = (\text{if } i = b_0 \lor i \in \mathit{lasts} \text{ then } 0 \text{ else } 1) \end{array}\right\}$$

```
  for (i=0; i<B; i++)
      if (available[i]) {
        writing=i; return i;
      }
  assert(0);
}
```

$\{\lambda b.\, \texttt{writing} \mapsto b * \texttt{last_given} \mapsto b_0 * \texttt{last_taken} \mapsto \mathit{lasts} \wedge b \notin \{b_0\} \cup \mathit{lasts}\}$

The `available` array computes the availability predicate $b \neq b_0 \wedge b \notin \mathit{lasts}$ for each buffer ID b; the final loop then searches for some b for which the predicate holds. We know that at least one such b exists (and thus the assertion will never be reached) by the pigeonhole principle: there is one value of b_0, N entries in `last_taken`, and $B = N + 2$ possible buffer IDs, so at least one buffer ID must be available. □

Theorem A.7. *The* `finish_write` *function meets its specification.*

Proof.

$$\left\{\begin{array}{l} \texttt{writing} \mapsto b * \texttt{last_given} \mapsto b_0 * \texttt{last_taken} \mapsto \mathit{lasts} * \\ \circledast_{\mathsf{r}} \left(\texttt{comm[r]} \odot\!\!\xrightarrow{.5} (g[\mathsf{r}], 0, R_{\mathsf{r}}, h[\mathsf{r}]) * g_1[\mathsf{r}] \overset{.5}{\mapsto} b_0 * g_2[\mathsf{r}] \overset{.5}{\mapsto} \mathit{lasts}[\mathsf{r}]\right) * \\ \circledast_{b'} \texttt{bufs}[b'] \xmapsto{\pi_b(b_0, \mathit{lasts})} _ \wedge b_0 \notin \mathit{lasts} \wedge b \notin \{b_0\} \cup \mathit{lasts} \end{array}\right\}$$

```
void finish_write(){
  for (r=0; r<N; r++) {
    b = atomic_exchange (&comm[r], writing);
    if (b==Empty)
       last_taken[r] = last_given;
  }
  last_given = writing;
  writing = Empty;
}
```

$$\left\{\begin{array}{l} \texttt{writing} \mapsto \texttt{Empty} * \texttt{last_given} \mapsto b * \exists \mathit{lasts}'.\, \texttt{last_taken} \mapsto \mathit{lasts}' * \\ \circledast_{\mathsf{r}} \left(\begin{array}{l} \exists \text{fresh } t'.\, \exists v'.\, \texttt{comm[r]} \odot\!\!\xrightarrow{.5} (g[\mathsf{r}], 0, R_{\mathsf{r}}, h[\mathsf{r}] \cup \{(t', [v' \to b])\}) * \\ g_1[\mathsf{r}] \overset{.5}{\mapsto} b * g_2[\mathsf{r}] \overset{.5}{\mapsto} \mathit{lasts}'[\mathsf{r}] \end{array}\right) * \\ \circledast_{b'} \texttt{bufs}[b'] \xmapsto{\pi_{b'}(b, \mathit{lasts}')} _ \wedge b \notin \mathit{lasts}' \end{array}\right\}$$

For each reader, the writer publishes the corresponding share of the written buffer b and checks for an acknowledgement. As in `start_read`, there are two cases for each location buffer `comm[r]`: either its value is `Empty` and the data buffer contained is the second most recently read buffer, or its value is some non`Empty` v and the buffer contained is v. In the former case, the previously written buffer ID b_0 has been received by `r`, and the ghost variables guarantee that the ID of the returned buffer is `last_taken[r]`. In this case, the writer also updates the value of `last_taken[r]`. In the latter case, the ghost variables guarantee that value v can only be the previously written buffer ID b_0, which the writer recollects. In this case, `r` cannot have taken any buffers since the last check, so `last_taken` is not updated. The loop invariant indicates the gradual construction of the new `last_taken` list lasts' in parallel with the associated share exchanges, maintaining the relationship between the two. □

A.3 A Verified Client (Section 6.1.5)

An `initialize_channels` function sets up the initial state of the system, creating the locks, setting the virtual memory permissions, and distributing the starting shares. After that, each client is a loop (here infinite) that uses the messaging functions as an interface to the buffers.

A.3.1 Reader Client.

$$\left\{\begin{array}{l}\texttt{reading[r]} \mapsto _ * \texttt{last_read[r]} \mapsto b_0 * \texttt{comm[r]} \odot\!\xrightarrow{.5} (g[\texttt{r}], 0, R_{\texttt{r}}, \texttt{nil}) * \texttt{bufs[}b_0\texttt{]} \overset{\pi_{\texttt{r}}}{\longmapsto} _ * \\ g_0[\texttt{r}] \overset{.5}{\longmapsto} b_0\end{array}\right\}$$

```
void reader(int r){
  while(1){
    buf_id b = start_read(r);
```

$$\left\{\begin{array}{l}\texttt{reading[r]} \mapsto b * \texttt{last_read[r]} \mapsto b * \texttt{comm[r]} \odot\!\xrightarrow{.5} (g[\texttt{r}], 0, R_{\texttt{r}}, h) * \texttt{bufs[}b\texttt{]} \overset{\pi_{\texttt{r}}}{\longmapsto} _ * \\ g_0[\texttt{r}] \overset{.5}{\longmapsto} b\end{array}\right\}$$

```
    buffer *buf = bufs[b];
    int v = buf->data;
    finish_read(r);
  }
}
```

After the call to `start_read`, the reader holds a share of the returned buffer b, allowing it to read from `bufs[b]`. The precondition of `start_read` is maintained as a loop invariant.

A.3.2 Writer Client.

$$\left\{\begin{array}{l}\texttt{writing} \mapsto _ * \texttt{last_given} \mapsto b_0 * \texttt{last_taken} \mapsto \mathit{lasts} * \\ \circledast_{\texttt{r}} \left(\texttt{comm[r]} \odot\!\xrightarrow{.5} (g[\texttt{r}], 0, R_{\texttt{r}}, \texttt{nil}) * g_1[\texttt{r}] \overset{.5}{\longmapsto} b_0 * g_2[\texttt{r}] \overset{.5}{\longmapsto} \mathit{lasts}[\texttt{r}]\right) * \\ \circledast_b \texttt{bufs[}b\texttt{]} \xmapsto{\pi_b(b_0, \mathit{lasts})} _ \wedge b_0 \notin \mathit{lasts}\end{array}\right\}$$

```
void writer(){
  int v = 0;
  while(1){
    buf_id b = start_write();
```

$$\left\{\begin{array}{l}\texttt{writing} \mapsto b * \texttt{last_given} \mapsto b_0 * \texttt{last_taken} \mapsto \mathit{lasts} * \\ \circledast_{\texttt{r}} \left(\texttt{comm[r]} \odot\!\xrightarrow{.5} (g[\texttt{r}], 0, R_{\texttt{r}}, h[r]) * g_1[\texttt{r}] \overset{.5}{\longmapsto} b_0 * g_2[\texttt{r}] \overset{.5}{\longmapsto} \mathit{lasts}[\texttt{r}]\right) * \\ \circledast_{b'} \texttt{bufs[}b'\texttt{]} \xmapsto{\pi_{b'}(b_0, \mathit{lasts})} _ \wedge b_0 \notin \mathit{lasts} \wedge b \notin \{b_0\} \cup \mathit{lasts}\end{array}\right\}$$

$$\left\{\begin{array}{l}\texttt{writing} \mapsto b * \texttt{last_given} \mapsto b_0 * \texttt{last_taken} \mapsto \mathit{lasts} * \\ \circledast_{\texttt{r}} \left(\texttt{comm[r]} \odot\!\xrightarrow{.5} (g[\texttt{r}], 0, R_{\texttt{r}}, h[r]) * g_1[\texttt{r}] \overset{.5}{\longmapsto} b_0 * g_2[\texttt{r}] \overset{.5}{\longmapsto} \mathit{lasts}[\texttt{r}]\right) * \\ \texttt{bufs[b]} \mapsto _ * \circledast_{b' \neq b} \texttt{bufs[}b'\texttt{]} \xmapsto{\pi_{b'}(b_0, \mathit{lasts})} _ \wedge b_0 \notin \mathit{lasts} \wedge b \notin \{b_0\} \cup \mathit{lasts}\end{array}\right\}$$

```
    buffer *buf = bufs[b];
    buf->data = v;
    finish_write();
```

$$\left\{\begin{array}{l}\texttt{writing} \mapsto \texttt{Empty} * \texttt{last_given} \mapsto b * \texttt{last_taken} \mapsto \mathit{lasts}' * \\ \circledast_{\texttt{r}} \left(\texttt{comm[r]} \overset{.5}{\odot\!\!\longrightarrow} (g[\texttt{r}], 0, R_{\texttt{r}}, h'[r]) * g_1[\texttt{r}] \overset{.5}{\longmapsto} b * g_2[\texttt{r}] \overset{.5}{\longmapsto} \mathit{lasts}'[\texttt{r}]\right) * \\ \circledast_{b'} \texttt{bufs}[b'] \xmapsto{\pi_b(b, \mathit{lasts}')} _ \wedge b \notin \mathit{lasts}'\end{array}\right\}$$

```
    v++;
  }
}
```

The call to `start_write` returns an available buffer b, i.e., one that is not in $\{b_0\} \cup \mathit{lasts}$. This means that the writer share $\pi_b(b_0, \mathit{lasts})$ is the full share, allowing the writer to write to `bufs[b]`. The invariant of `start_write` is reestablished by `finish_write` (with a new collection of histories), so the loop invariant holds.