Verifying Read-Copy-Update in a Logic for Weak Memory

Joseph Tassarotti
Carnegie Mellon University, USA
jtassaro@andrew.cmu.edu

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

Viktor Vafeiadis
MPI-SWS, Germany
viktor@mpi-sws.org

Abstract

Read-Copy-Update (RCU) is a technique for letting multiple readers safely access a data structure while a writer concurrently modifies it. It is used heavily in the Linux kernel in situations where fast reads are important and writes are infrequent. Optimized implementations rely only on the weaker memory orderings provided by modern hardware, avoiding the need for expensive synchronization instructions (such as memory barriers) as much as possible.

Using GPS, a recently developed program logic for the C/C++11 memory model, we verify an implementation of RCU for a singly-linked list assuming "release-acquire" semantics. Although release-acquire synchronization is stronger than what is required by real RCU implementations, it is nonetheless significantly weaker than the assumption of sequential consistency made in prior work on RCU verification. Ours is the first formal proof of correctness for an implementation of RCU under a weak memory model.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory, Verification

Keywords Concurrency; Weak memory models; C/C++; RCU; Program logic; Separation logic

1. Introduction

Traditionally, most work on concurrent program verification has assumed a *sequentially consistent (SC)* model of memory, in which updates to memory are globally visible to all threads as soon as they occur [10]. For performance reasons, however, modern architectures offer weaker guarantees about the ordering of concurrent memory operations [11, 16]. Although it is possible to simulate SC semantics on such hardware by inserting explicit synchronization instructions (*e.g.,* barriers/fences), the cost of doing so—particularly for high-performance concurrent code—can be prohibitive.

Fortunately, for many concurrent algorithms, full SC behavior is unnecessary, and more limited forms of synchronization suffice. One widely-used example is Read-Copy-Update (RCU) [12, 13]. RCU is a technique, deployed heavily in the Linux kernel, that lets a single writer manipulate a data structure while multiple readers are concurrently accessing it. Instead of directly modifying a piece of the structure, the writer first copies that piece, modifies the copy, and then makes the new copy accessible and the old one inaccessible. Once no readers are capable of accessing the old copy, the writer may safely deallocate it. However, until that time, some readers may see the old copy while others see the new copy, and there is no guarantee when readers will begin to see the new copy.

As this description suggests, RCU employs *some* synchronization (*e.g.,* to ensure memory safety), but *not* full SC semantics, and its reliance on weaker memory assumptions is essential to its efficiency. However, the only existing formal proof of correctness for an RCU-based data structure [7] assumes an SC memory model.

In this paper, we give the first formal proof of correctness for an implementation of RCU under a weak memory model. Specifically, we verify a user-space RCU implementation of linked lists (based on that of Desnoyers et al. [5]), programmed using *release-acquire atomics*, one of the main weak-memory access modes supported by the C/C++11 language standard [8].

Why focus on release-acquire? There are several reasons. First, the semantics of C11's release-acquire mode has been fully formalized [3], rendering our RCU implementation amenable to formal verification, and unlike several other features of the C11 model [20], its semantics is relatively uncontroversial. Second, release-acquire semantics, while significantly weaker than SC, nevertheless provides sufficiently strong synchronization to guarantee safety of our RCU implementation. In fact, release-acquire provides stronger semantics than what real RCU implementations require—the release-consume mode of C11 was designed specifically to support RCU, but the "right" semantics of release-consume is still a matter of debate, and at present most C compilers do not implement it differently from release-acquire [14]. Third, release-acquire semantics is "reasonable", in the sense that one can reason about it using a more restricted version of the kinds of reasoning principles that hold under SC semantics. This claim was substantiated formally by recent work of Turon et al. [17] on a logic called GPS, which supports Hoare-style verification of C11 programs under release-acquire semantics. Here, we leverage (a mild extension of) GPS in our verification, while simultaneously demonstrating a much more significant case study for the use of GPS than any that Turon et al. previously considered.

Above and beyond our formal verification of RCU in GPS (described in full formal detail in our technical appendix [1]), an important contribution of this work is the proof *idea* itself, and it is the elucidation of this proof idea that is our main goal in this paper. Previously, the correctness of RCU has been argued using the concept of a *grace period* during which reader threads may finish accessing an old node before it is deallocated. Indeed, Gotsman et al.'s proof in the SC setting depends on an extension of separation logic with a temporal "since" operator to formalize grace periods. In contrast, our proof avoids any such extension; we rely instead on GPS's notion of *per-location protocols*, which describe how the state of a shared memory location may evolve over time. Using per-location protocols, Turon et al. showed how to formalize

PLDI'15, June 13–17, 2015, Portland, OR, USA.

ACM 978-1-4503-3468-6/15/06…$15.00.
http://dx.doi.org/10.1145/2737924.2737992

the folklore intuition that release-writes and acquire-reads support a form of *message passing* between threads. We in turn show how such message passing provides a self-sufficient explanation of how RCU works—*e.g.,* of how the writer tells the readers which nodes have been deallocated, or how the readers tell the writer that they will no longer access them. No additional mechanism is required.

In the rest of the paper, we present the semantics of release-acquire (§2), our implementation of RCU and why intuitively it works (§3), our specification of RCU and why it is useful (§4), a review of GPS with some minor extensions (§5), a high-level description of our verification of RCU in GPS (§6), and discussion of related work (§7).

2. Release-Acquire Semantics

Our implementation of RCU uses a simple imperative while-language extended with C11's release-acquire memory operations. In this section, we try to give some intuition for the semantics of these operations. Thorough, formal presentations of the C11 memory model can be found in Batty et al. [3] and Vafeiadis et al. [20].

The C11 standard divides memory accesses into two types—*atomic* and *non-atomic*—and we will say that a memory location is atomic (resp. non-atomic) if all accesses to it are of that type. Non-atomic accesses are the default variety, appropriate for most use cases: they are fast and do not require the compiler to emit special synchronization instructions, but the standard says that the behavior of a program is undefined if it involves racy non-atomic accesses (*i.e.,* two unordered non-atomic accesses to a single location, at least one of which is a write). Thus, the onus is on the programmer to employ enough synchronization to ensure that there are no non-atomic data races.

One way of implementing such synchronization is using atomic locations, which are *intended* for racy access. The C11 standard lets programmers annotate reads and writes for atomic locations with different consistency level options, ranging from sequentially consistent (SC) to fully relaxed, depending on how cheap they want the accesses to be and how much instruction reordering they can tolerate.

We focus here on only two of these options: *release writes* and *acquire reads*. When a thread performs a release write, and another thread observes that write via an acquire read, the operations that precede the write are guaranteed to "happen-before" the read. This kind of "transitive visibility", as it is often called, allows programmers to use release-acquire to implement a message-passing idiom.

For example, consider the following contrived yet illustrative code (which we will show how to verify in §5):

```
[x]na := 37;                  ||  [y]na := 25;
[lf]rel := 1;                 ||  if ([lf]acq == 1) {
while ([rf]acq != 1) {        ||    [y]na := [x]na;
  /* spin */                  ||  }
}                             ||  [rf]rel := 1;
[x]na := 49;                  ||  /* postcond: y ↪ 37 ∨ y ↪ 25 */
```

Here, we have two non-atomic memory locations, x and y, and two atomic "flags", lf and rf. We write $[x]_{\mathtt{na}}$ to indicate a non-atomic operation on location x, and $[lf]_{\mathtt{rel}}$ and $[lf]_{\mathtt{acq}}$ for release and acquire operations on lf. The code on the sides of the vertical bars represents two different threads, and initially we assume that all locations contain 0.

The effect of this code is to pass control of the non-atomic location x from the left thread to the right thread and back. The left thread initializes location x with value 37, and then sets its flag lf to 1 with a release write, to signal that the right thread can now access x. The right thread checks if lf is set to 1, and if so, it reads the value of x (non-atomically) and sets y to that value; otherwise, it sets y to 25. Because the non-atomic write of 37 to x preceded the release write to lf, and the matching acquire read of lf as 1 (if it occurred) preceded the non-atomic read of x, we will be able to establish: (1) the write to x happened before the read of x, so there is no data race on x, and (2) as a postcondition, y may either point to 25 or 37, but not 0. After the second thread finishes accessing x, it does a release write of 1 to rf to signal to the left thread that it is done. The first thread spins until it observes this write. At this point, it knows it can safely update x again because the second thread's read of x must have happened already.

In the previous example, the two threads synchronized their operations via release writes to the atomic locations lf and rf to send messages back and forth, and acquire reads to receive them. In contrast, here is an example based on the classic "Dekker's algorithm" for mutual exclusion [6], which is safe under SC but *not* under release-acquire:

```
[x]rel := 1                   ||  [y]rel := 1
if [y]acq == 0 then           ||  if [x]acq == 0 then
  [z]na := 1                  ||    [z]na := 2
```

Here, the left thread does a release write to set x to 1, while the right thread does the same for y. Each thread then does an acquire read of the other's variable to see if it has been updated. If not, each thread tries to modify the non-atomic z.

Under an SC semantics, one of the thread's writes must happen before the other: that thread "wins" and may write to z. For instance, if the left thread reads y and sees 0, it concludes that the right thread has not written to y yet, so the left thread knows it has won the race. However, under release-acquire, the writes to x and y are unordered, and it is possible for the left thread to read 0 from y *and* for the right thread to read 0 from x in the same execution. If this happens, they will *both* try to write to z, resulting in a data race.

Informally, if we think in terms of message passing, this example is unsound because it tries to conclude something from the negative fact that a message has *not* yet arrived. To be safe, under release-acquire, we can only draw sound conclusions from the positive information that a message *has* arrived, as in the first example. Later in §5, we will see how this intuitive reasoning is formalized in GPS.

3. RCU

We now describe how RCU can be implemented for a singly linked list using the release-acquire memory operations. In explaining the algorithm, we focus on how the orderings imposed by pairs of release-acquire operations ensure that there are no data races. In each case, we can informally describe these operations in terms of how they send messages between the threads. In §6 our proof will make this message-passing explanation precise.

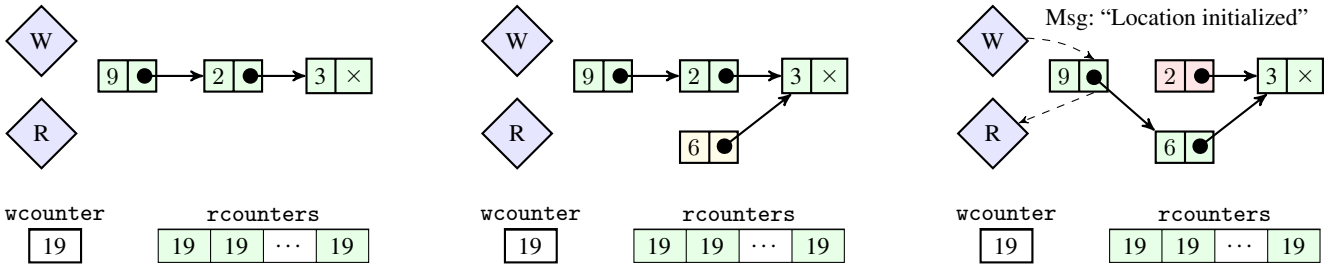
A simplified part of our verified implementation is presented in Figure 1. Nodes in the list are records with two fields. The `data` field contains the contents of the node, and `link` is a pointer to the next node in the list.

Initialization A new RCU instance is created by calling `rcuNew`. This returns a pointer q to the metadata for the RCU instance, which consists of a counter for the writer (q + `wcounter`), an array of counters for the readers (q+`rcounters`, which we describe below), a field containing a pointer to the head of the list (q + `link`), and a pointer to a structure used for a custom allocator that recycles deallocated nodes (q + `free`). The counters all start at 0, and the q + `link` field is initially a null pointer.

Reading The readers access the nodes in the list in a loop that maintains a current pointer into the list. They start the traversal by calling `rcuReadStart` to get the first pointer to the head of the list. This does an acquire read on q + `link` and returns the result.

```
    rcuNew() ≜
1:    let q = alloc(N + 3)
2:    [q + link]_rel := 0;
      for i = 0 to N − 1 do
3:        [q + rcounters + i]_rel := 0;
4:    [q + wcounter]_rel := 0;
5:    [q + free]_na := rcuAllocInit();
6:    q

    rcuReadStart(q) ≜
7:    [q + link]_acq

    rcuReadNext(q, p) ≜
8:    let v = [p + data]_na
9:    let p' = [p + link]_acq
10:   (v, p')

    rcuNodeAppend(q, p, v) ≜
11:   let x = rcuAlloc(q)
12:   [x + data]_na := v;
13:   [x + link]_rel := 0;
14:   [p + link]_rel := x;
15:   x

    rcuNodeUpdate(q, x, p, v) ≜
16:   let c = [x + link]_acq
17:   let x' = rcuAlloc(q)
18:   [x' + data]_na := v;
19:   [x' + link]_rel := c;
20:   [p + link]_rel := x';
21:   rcuSynchronize(q);
22:   rcuFree(q, x);
23:   x'

    rcuNodeDelete(q, x, p) ≜
24:   let c = [x + link]_acq
25:   [p + link]_rel := c;
26:   rcuSynchronize(q);
27:   rcuFree(q, x);

    rcuSynchronize(q) ≜
28:   let oldgc = [q + wcounter]_acq
29:   let newgc = oldgc + 1
30:   [q + wcounter]_rel := newgc;
31:   for i = 0 to N − 1 do
32:     while [q + rcounters + i]_acq != newgc;
33:   end

    rcuQuiescentState(q, tid) ≜
34:   let t = [q + wcounter]_acq
35:   [q + rcounters + tid]_rel := t
```

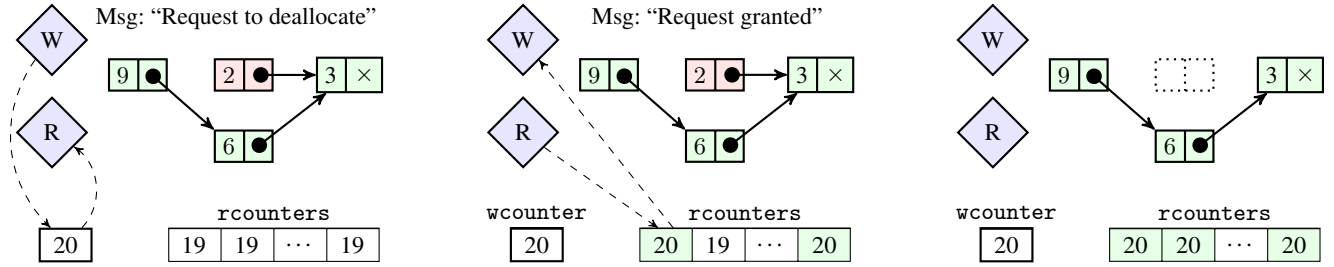
Figure 1. A concurrent linked list with a single writer implemented using QSBR RCU.



(a) State before `rcuNodeUpdate`.

(b) State after the writer executed line 19.

(c) State after the writer linked the new node onto the list (line 20).



(d) State after the writer updated its counter (line 30).

(e) State after the reader has updated its counter (line 35) and the writer noticed it (line 32).

(f) State after `rcuSynchronize`(q) finished. The writer can safely free the node (line 22).

Figure 2. Illustration of updating a node in the list using RCU.

Then, within a loop, they check that their current pointer is not null, and if so, access the value stored at the node and the next node by calling `rcuReadNext` with their current pointer as p. The function simply reads off the `data` and `link` fields of p and returns the results. While the `data` field is read non-atomically, the `link` field is accessed by an acquire read—this ensures correct synchronization with the writer.

Updating the list We now explain how the writer modifies the list by walking through an example shown in Figure 2. It depicts the linked list, as well as the counters in the RCU metadata. The writer thread and one reader thread are represented by diamonds labeled "W" and "R". To represent messages being passed from one thread to the other through a location, we draw a dashed arrow from the sender to the location and another from the location to the reader.

Suppose the writer wants to change the value in the second node of the linked list shown in Figure 2a. To do so, it calls `rcuNodeUpdate`(q, x, p, v), where q is a pointer to the RCU metadata, x is a pointer to the node that it wants to modify, p is a pointer to the previous node, and v is the new value for the node. This function first allocates a new node by calling `rcuAlloc`(q) (not shown in the figure). Next, the writer copies the old node's `link` field, and sets the updated value as in Figure 2b. Then, it updates the previous node's `link` field with a release write so that it points to the new node (line 20), rendering the old node unreachable as shown in 2c. At this point, readers may begin to see the new node when they do an acquire read on the previous `link` pointer (line 9). The pairing between this release write and the readers' acquire reads is the first important point of synchronization in RCU:

Release-Acquire Pair 1 (`link` field): The ordering imposed by the release writes and acquire reads on the `link` fields ensures that the initialization of the node precedes the readers' accesses. In other words, the writer passes a message to the readers saying that the next node is safe to access.

Similar synchronization points occur when writers append new nodes onto the end of the list with `rcuNodeAppend` or delete a node from the list with `rcuNodeDelete`.

Deallocating the old node After completing the release write in Figure 2c, the writer calls `rcuSynchronize` to wait until no readers can access the old copy any longer, so that it can deallocate the removed node. There are a number of ways to implement `rcuSynchronize` without sacrificing reader performance. The code in Figure 1 uses Quiescent State Based Reclamation (QSBR) [5].

In QSBR, the writer begins the synchronization operation by incrementing its counter (line 30 and Figure 2d). It then repeatedly reads each reader counter in turn until they all match the writer counter's new value. When readers are not accessing the list (that is, they are *quiescent*), they periodically call `rcuQuiescentState`, which examines the writer's counter and copies its value into the reader's counter (Figure 2e). Once the writer sees that every reader's counter matches its own, it knows they have all entered a quiescent state since the old node became unreachable. This means that if the readers access the list in the future, they will not access the old node, so the node can be safely deallocated (Figure 2f).

These counter fields must be atomic, because the readers will try to concurrently read the writer's counter as the writer is incrementing it, and vice-versa. Both counters are involved in a release-acquire synchronization:

Release-Acquire Pair 2 (`wcounter` field): The synchronization between the write on line 30 and the read on line 34 guarantees that once the reader sees the updated counter, it will see the update that made the old node unreachable (line 20). When the writer increments its counter, it publishes the fact that a node has been made unreachable together with a request for permission to deallocate the node.

Release-Acquire Pair 3 (`rcounters` + tid field): When the readers update their counters to match the writer's on line 35, they acknowledge the writer's request by giving up their own permission to access the unreachable node. The release-acquire ordering ensures that any accesses the reader was doing before calling `rcuQuiescentState` all finish before the writer proceeds to deallocate the node.

Memory management After calling `rcuSynchronize`, the writer calls `rcuFree` (not shown in Figure 1; see the appendix [1]). Earlier work on formally specifying the C11 memory model [3] did not address the semantics of `free`. Since our focus is on RCU rather than the behavior of deallocation in C11, we have opted to remain within the scope of what we can formalize—we thus hand-roll our own naive, *ad hoc* memory reclamation routine in `rcuFree`. The call to `rcuFree` adds the address of the old node to a pool. The implementation of `rcuAlloc` (again, see the appendix) first tries to remove an address from the pool and return it. If the pool is empty, it calls `alloc`().

In this simplified version of our implementation, there is a fixed number of readers, N, and the writer immediately synchronizes and deallocates the old node as soon as it performs an update. In the full version verified in the appendix, we allow the readers to register themselves dynamically and let the writer batch its deallocations for efficiency.

$\{\mathsf{true}\}$
`rcuNew`()
$\left\{q.\, \exists H.\, \mathsf{WriterSafe}(q, [(q, \mathsf{null})]) * \mathop{\Large *}_{tid<N} \mathsf{ReaderSafe}(q, H, tid)\right\}$

$\{\mathsf{WriterSafe}(q, L \cdot (p, v) \cdot L')\}$
$[p + \texttt{link}]_{\texttt{acq}}$
$\left\{\begin{array}{l} p'.\, \mathsf{WriterSafe}(q, L \cdot (p, v) \cdot L') * ((p' = 0 \wedge L' = \mathtt{nil}) \\ \quad \vee (p' \neq 0 \wedge \exists L'', v'.\, L' = (p', v') \cdot L'')) \end{array}\right\}$

$\{\mathsf{WriterSafe}(q, L \cdot (p, v) \cdot L') \wedge v \neq \mathsf{null}\}$
$[p + \texttt{data}]_{\texttt{na}}$
$\{x.\, x = v \wedge \mathsf{WriterSafe}(q, L \cdot (p, v) \cdot L')\}$

$\{\mathsf{WriterSafe}(q, L \cdot (p, v)) * \square P(v')\}$
`rcuNodeAppend`(q, p, v')
$\{x.\, \mathsf{WriterSafe}(q, L \cdot (p, v) \cdot (x, v'))\}$

$\{\mathsf{WriterSafe}(q, L \cdot (p, v_0) \cdot (x, v_1) \cdot L') * \square P(v_1')\}$
`rcuNodeUpdate`(q, x, p, v_1')
$\{x'.\, \mathsf{WriterSafe}(q, L \cdot (p, v_0) \cdot (x', v_1') \cdot L')\}$

$\{\mathsf{WriterSafe}(q, L \cdot (p, v_0) \cdot (x, v_1) \cdot L')\}$
`rcuNodeDelete`(q, x, p)
$\{\mathsf{WriterSafe}(q, L \cdot (p, v_0) \cdot L')\}$

$\{\mathsf{ReaderSafe}(q, H, tid)\}$
`rcuReadStart`(q)
$\{p.\, \mathsf{ReaderSafe}(q, H, tid) * \square\mathsf{SafePtr}(q, H, p)\}$

$\{\mathsf{ReaderSafe}(q, H, tid) * \square\mathsf{SafePtr}(q, H, p) * p \neq 0\}$
`rcuReadNext`(q, p)
$\{(v, p').\, \mathsf{ReaderSafe}(q, H, tid) * \square\mathsf{SafePtr}(q, H, p') * \square P(v)\}$

$\{\mathsf{ReaderSafe}(q, -, tid)\}$
`rcuQuiescentState`(q, tid)
$\{\exists H'.\, \mathsf{ReaderSafe}(q, H', tid)\}$

Figure 3. Specifications of the RCU operations.

4. RCU Specification

GPS [17] lets us prove Hoare-style triples of the form:

$$\{P\}\ e\ \{x.\, Q\}$$

asserting that if a thread starts with the resources described by P and executes expression e, then:

- The execution of e is guaranteed to be free of memory errors (e.g. accessing uninitialized data) and non-atomic data races.
- If e terminates with value V, then $[V/x]Q$ describes the thread's resources afterward.

Later, we will review the logical mechanisms that GPS provides for proving these triples. For now, we describe the specification for RCU that we will prove. The full specification is shown in Figure 3. We assume some fixed predicate $P(x)$ that we require to hold of values stored in the list. Any value inserted by the writer must satisfy this predicate, and readers are guaranteed that values they get out will also satisfy it. The RCU specification then employs three predicates which are defined in terms of underlying GPS primitives but can be treated abstractly by a client: $\mathsf{WriterSafe}(q, L)$, $\mathsf{ReaderSafe}(q, H, tid)$, and $\mathsf{SafePtr}(H, p)$.

$\mathsf{WriterSafe}(q, L)$ represents the permissions owned by the writer. The logical list L is of the form $(q, \mathsf{null}) \cdot (l_1, v_1) \cdot$

$(l_2, v_2) \cdots (l_n, v_n)$, where l_i is a pointer to the ith node in the list, and v_i is the value stored in the `data` field of that node. The predicate says that for the RCU structure with metadata at q, the physical list contains the nodes mentioned in L. We generate this permission when we create a new RCU instance, at which point L consists only of (q, null). Accessing the `link` field of a pointer in L just returns the next pointer in L. Each of the writer's methods consumes this permission, and returns a version where the contents of L have been modified accordingly.

$\mathsf{ReaderSafe}(q, H, tid)$ is the analogous permission for readers. From the perspective of the client code, H is completely abstract: it simply represents the version of the list that the tid-th reader sees. $\mathsf{SafePtr}(q, H, p)$ means that p is a pointer to a properly initialized node. The specification for `rcuReadStart` says that it always returns a SafePtr. As the reader inspects the list using `rcuReadNext`, p must be a non-null SafePtr, and when the call returns, it returns another SafePtr.

When the reader calls $\texttt{rcuQuiescentState}(q)$, it gives up its current $\mathsf{ReaderSafe}(q, H, tid)$ and receives $\mathsf{ReaderSafe}(q, H', tid)$ in return, for some fresh H'. This makes any previous SafePtr assertions unusable, and forces the reader to start again at the head of the list by getting a new SafePtr from `rcuReadStart`.

Finally, note that some predicates (P and SafePtr) are "boxed", *i.e.,* appearing under a $\Box$ modality. This means that these predicates denote "duplicable facts" (with the property that $\Box Q \Leftrightarrow \Box Q * Q$) as opposed to uniquely owned permissions, a distinction that will be explored further in the next section.

5. GPS

In this section, we briefly review some of the key mechanisms in GPS that we will use in verifying our RCU implementation. We then illustrate their use on the simple message-passing example from §2. Although the example is contrived, its verification closely mirrors the structure of the RCU verification, and it shows off all the features of GPS working in tandem. It is thus quite useful as a warm-up for the main attraction.

5.1 Key Features of GPS

The four key features of GPS are as follows.

Ownership of non-atomics The assertion $x \hookrightarrow v$ says that x is a non-atomic location pointing to the value v. This assertion is precisely the standard points-to assertion of separation logic [15]: whoever asserts $x \hookrightarrow v$ is the exclusive "owner" of x, and has the freedom to read and write it arbitrarily.

Here, we also extend GPS slightly to support *fractional permissions* [4] on non-atomic locations. We annotate the points-to relation with a permission k, which is an element of a permission algebra [18]. This algebra is a set with a distinguished element $\top$, representing "full" permission, and a partial operation $\oplus$ for combining permissions. Now, $x \overset{k}{\hookrightarrow} v$, where $k \neq \top$, denotes only ownership of a partial permission to access x, which means the ability to read x but not write it. The initial (full) owner of x may thus split up its ownership assertion into pieces to be given out to readers, and then later on collect those pieces to reconstitute the full permission so that it can update x. Crucially, though, with neither full nor fractional ownership is it possible for one thread to read x at the same time another may be writing it: thus, we guarantee absence of data races on non-atomics.

In the RCU proof, since we assume one writer and a fixed number of readers, N, our permission algebra will be sets of thread IDs, with $\top = \{0, \ldots, N\}$ and $\oplus$ defined as disjoint set union. We write $x \overset{tid}{\hookrightarrow} v$ (for $0 \leq tid \leq N$) as shorthand for $x \overset{\{tid\}}{\hookrightarrow} v$, the partial permission for thread tid to read x (thread N is the writer).

Protocols for message passing via atomics Unlike non-atomics, atomic locations are meant to be read and written simultaneously. We therefore cannot make any stable assertions about the precise contents of an atomic location, but we *can* assert something about how those contents are permitted to evolve over time. We call such an assertion a *protocol* assertion, $\boxed{x : s \mid \tau}$. It asserts two things. First, it says that x is governed by the protocol τ. This protocol consists of a partially ordered set of logical *states* S that x can be in, together with an *interpretation* function $\tau(s, v)$ that says what assertion must hold when x is in logical state $s \in S$ and stores value v. Second, the protocol assertion says that x is *at least* in state s of its protocol. This assertion is a duplicable fact, and may thus be shared freely between threads, because GPS requires writes to x to always *advance* the state of its protocol—so once x is at least in state s, it will remain so forever.

Through their interpretation functions, protocols offer a way for threads to pass messages to each other. Specifically, suppose two threads both know $\boxed{x : s \mid \tau}$. When one of the threads writes v to x, it must be able to prove that $\tau(s', v)$ holds for some future state s' of s. Subsequently, when the other thread performs a read on x, observing value v, it will learn that there is some future state s' of s such that $\tau(s', v)$ holds. The protocol has thus served to transmit the knowledge of $\exists s' \sqsupseteq s.\ \tau(s', v)$ from one thread to the other.

Exchanges for ownership transfer While protocols support the transfer of knowledge (*i.e.,* duplicable facts) between threads, *exchanges* support the transfer of *exclusive ownership* of resources between them.[1] This will be very important when verifying our message-passing example (see §5.2 below), wherein we want to pass exclusive ownership of $x \hookrightarrow 37$ back and forth between the two threads.

The exchange mechanism is very simple. Suppose P and Q are assertions such that $P * P \Rightarrow \mathsf{false}$ and $Q * Q \Rightarrow \mathsf{false}$, *i.e.,* they denote exclusive ownership, so two threads cannot assert P simultaneously (and likewise for Q). We write $\sigma : P \leftrightsquigarrow Q$ to say that σ is the name of an exchange between P and Q, and we write $\mathsf{exch}(\sigma)$ to represent the assertion that the exchange σ has been created. The idea is that σ, once created, represents an invariant governing some shared state, which asserts that that shared state *either* satisfies P or it satisfies Q. Once created, the σ invariant is enforced permanently, and thus the assertion $\mathsf{exch}(\sigma)$ is duplicable knowledge that can be freely shared amongst threads.

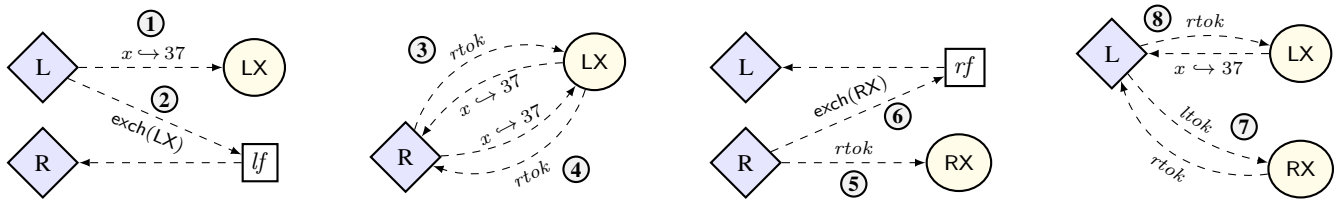
To see how exchanges support ownership transfer, suppose thread 1 owns P, thread 2 owns Q, and thread 1 wishes to transfer ownership of P to thread 2. Thread 1 can create the exchange σ by giving up ownership of P to the exchange, thereby learning $\mathsf{exch}(\sigma)$ in return. It may then use release-acquire message passing (as described above) to inform thread 2 of the knowledge that σ exists. Since thread 2 owns Q, it can then give up Q in exchange for P. These logical ownership transfers are summarized as follows:

$$(P \vee Q) \Rrightarrow \mathsf{exch}(\sigma) \quad P \wedge \mathsf{exch}(\sigma) \Rrightarrow Q \quad Q \wedge \mathsf{exch}(\sigma) \Rrightarrow P$$

Note that the assumption that assertions P and Q are exclusive (non-duplicable) is essential in order to ensure that there is a *unique* recipient of the ownership transfer. For instance, if Q were some duplicable fact, then multiple threads would be able to exchange Q for P, which would result (unsoundly) in multiple threads gaining simultaneous ownership of P.

Ghost PCMs for encoding auxiliary state Ghost (or auxiliary) state is a ubiquitous mechanism in program logics, enabling the verifier to record and manipulate additional *logical* state beyond the physical state manipulated by the program itself. GPS supports a

[1] The original version of GPS featured a slightly more limited primitive called *escrows*. Exchanges generalize escrows to support bidirectional transfer.



(a) The left thread creates and publishes the exchange LX.

(b) The right thread uses the LX exchange twice.

(c) The right thread creates and publishes RX.

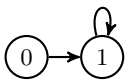
(d) The left thread uses LX and RX to get $x \hookrightarrow 37$.

Figure 4. Message passing and ownership transfer in the simple message-passing example.

$\mathsf{LX} : x \hookrightarrow 37 \leftrightsquigarrow \boxed{rtok : \diamond\ \mathsf{Tok}}$

$\mathsf{RX} : \boxed{rtok : \diamond\ \mathsf{Tok}} \leftrightsquigarrow \boxed{ltok : \diamond\ \mathsf{Tok}}$

lf, rf:



$\mathsf{LFP}(s, v) \triangleq (s = 0 * v = 0) \vee (s = 1 * v = 1 * \mathsf{exch}(\mathsf{LX}))$

$\mathsf{RFP}(s, v) \triangleq (s = 0 * v = 0) \vee (s = 1 * v = 1 * \mathsf{exch}(\mathsf{RX}))$

Figure 5. Protocols/exchanges for message-passing example.

very general notion of ghost state in the form of user-defined *partial commutative monoids (PCMs)*. Given a PCM μ and an element $t \in \mu$ we write $\boxed{\gamma : t\ \mu}$ to say that a thread owns element t from instance γ of the monoid μ. Then, we can split or combine ghost state elements as follows:

$$\boxed{\gamma : t \cdot_\mu t'\ \mu} \quad \Leftrightarrow \quad \boxed{\gamma : t\ \mu} * \boxed{\gamma : t'\ \mu}$$

In particular, if $t \cdot_\mu t'$ is not defined (which is possible since the monoid is partial), then $\boxed{\gamma : t\ \mu} * \boxed{\gamma : t'\ \mu} \Rightarrow \mathsf{false}$.

Recent work has shown that PCMs are remarkably expressive [9]. In this paper, we focus on two relatively simple types of PCMs that are relevant to the RCU proof: the *permission token* and *master/snapshot* PCMs.

Permission tokens represent capabilities to perform certain operations. We will use them, for instance, to represent the permissions to make certain steps in a protocol, or the permission to access certain exchanges. Permission tokens are defined by the monoid Tok which has two elements $\diamond$ and ϵ, with ϵ as identity and $\diamond \cdot \diamond$ undefined. The key property of a permission token is that it is non-duplicable ($\boxed{\gamma : \diamond\ \mathsf{Tok}} * \boxed{\gamma : \diamond\ \mathsf{Tok}} \Rightarrow \mathsf{false}$) and thus represents an exclusive capability. We discuss master/snapshot PCMs in §6.1.

5.2 Verifying the Message-Passing Example

To show how the above mechanisms work together, let us return to the first example in §2. Our verification, which we describe here at a high (but still detailed) level, guarantees two things: (a) the postcondition, namely that, once the threads terminate, y points to 25 or 37, and (b) that the code is "safe", meaning that there are no data races on the non-atomic x and y.

```
[x]na := 37;                    ||  [y]na := 25;
[lf]rel := 1;                   ||  if ([lf]acq == 1) {
while ([rf]acq != 1) {          ||    [y]na := [x]na;
  /* spin */                    ||  }
}                               ||  [rf]rel := 1;
[x]na := 49;                    ||  /* postcond: y ↪ 37 ∨ y ↪ 25 */
```

We walk through the proof now step by step. These steps are illustrated pictorially in Figure 4, and they involve protocols and exchanges that are defined formally in Figure 5.

At the start of the proof, we associate the flags lf and rf with the left and right flag protocols **LFP** and **RFP**, respectively (explained below). We also create the left and right permission tokens, $\boxed{ltok : \diamond\ \mathsf{Tok}}$ and $\boxed{rtok : \diamond\ \mathsf{Tok}}$, and give the left and right threads exclusive ownership of their respective tokens.

Step 1 (Fig. 4a): The left thread first sets x to 37. It then wants to transfer ownership of x to the right thread. To do so, it creates the exchange LX. By giving up ownership of $x \hookrightarrow 37$ to the exchange, it gains the knowledge exch(LX) that LX exists.

Step 2 (Fig. 4a): The left thread now wants to send its knowledge of exch(LX) to the right thread by setting its flag, lf, to 1. To reason about this, we use the left flag protocol **LFP**. This protocol asserts that x is initially 0, and that it may be set to 1 but can never be set back to 0 again. It also asserts that when lf is set to 1, it must be the case that exch(LX) holds. Since the left thread knows exch(LX), it is free to update lf to 1 (updating the logical state s of lf's **LFP** protocol to 1 as well).

Step 3 (Fig. 4b): The right thread may or may not observe that lf has been set to 1. In case it does not observe it, this and the next step are skipped. In case it *does* observe it, it learns that the **LFP** protocol must be in the 1 state, and hence it learns that LX exists. It then uses LX to exchange its own permission token, $rtok$, for ownership of $x \hookrightarrow 37$. Now that it owns x, it can safely read it and be sure that it will see the value 37.

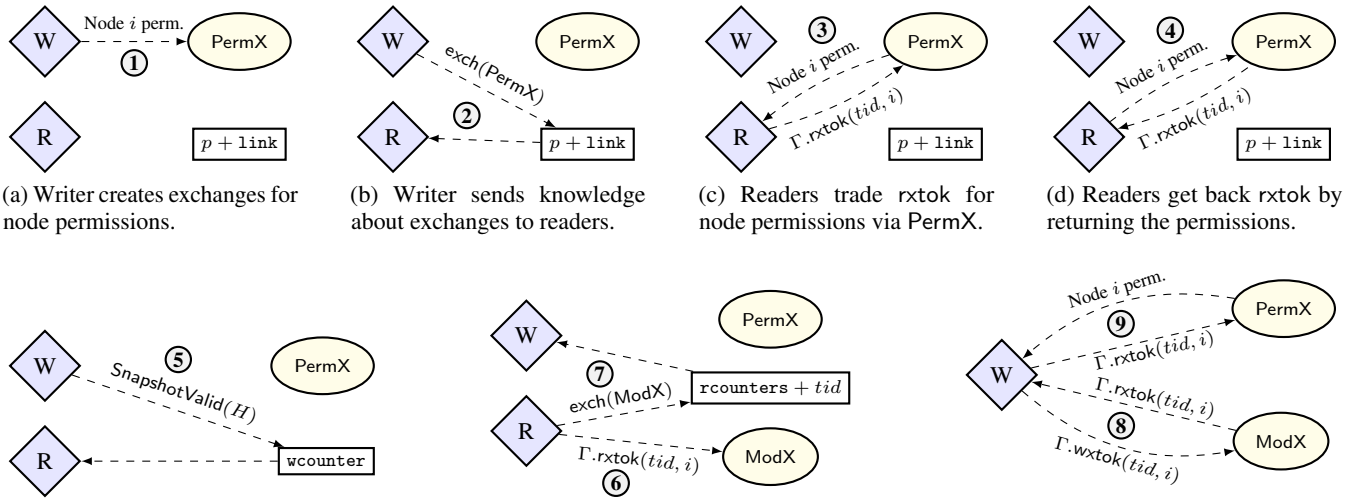
Step 4 (Fig. 4b): The right thread now wants to transfer ownership of x back to the left thread. To achieve this, its first step is to perform the reverse trade on LX, putting ownership of $x \hookrightarrow 37$ back under control of LX in return for its permission token $rtok$. **Note**: at this point, regardless of whether Steps 3 and 4 were performed or not, y points to either 25 or 37, as desired.

Step 5 (Fig. 4c): The right thread next creates the exchange RX by transferring its permission token $rtok$ into the exchange. In doing so, it learns exch(RX).

Step 6 (Fig. 4c): The right thread now wants to send its knowledge of exch(RX) to the left thread by setting its flag, rf, to 1. To reason about this, we use a right flag protocol, **RFP**, that is very similar to the left flag protocol, **LFP**, the only difference being that in state 1, **RFP** asserts exch(RX) (rather than exch(LX)). The right thread may thus set rf to 1 because it knows RX exists.

Step 7 (Fig. 4d): The left thread loops until it observes that rf has been set to 1. Once it observes this, it knows that the **RFP** protocol must be in state 1 and thus that RX exists. It then uses RX to exchange its own permission token, $ltok$, for the right permission token, $rtok$.

Step 8 (Fig. 4d): Finally, the left thread uses its original LX exchange to trade the right permission token, $rtok$, for ownership of $x \hookrightarrow 37$. It now knows that it has exclusive ownership of x and may therefore safely modify it again.



(a) Writer creates exchanges for node permissions.

(b) Writer sends knowledge about exchanges to readers.

(c) Readers trade rxtok for node permissions via PermX.

(d) Readers get back rxtok by returning the permissions.

(e) Writer requests deallocation permission by sending the list snapshot.

(f) Readers create exchange for requested rxtok and notify the writer.

(g) Writer uses ModX and PermX to retrieve the node permissions.

Figure 6. Message passing and ownership transfer in our RCU implementation.

6. RCU Proof Overview

In §5, we saw how protocols and exchanges could be used to implement message passing and ownership transfer. We now use these mechanisms to formalize the intuitive explanation we gave for RCU in §3.

There are two important ownership transfers involved in RCU: the writer transfers fractional ownership of nodes to readers when the nodes are added to the list, and readers transfer the fractional ownership of deleted nodes back to the writer during synchronization. The astute reader may note, however, that these nodes consist of both a non-atomic (`data`) field and an atomic (`link`) field, but in §5 we only discussed fractional ownership of non-atomics. Indeed, what does it even mean to "own" an atomic location, given that protocol assertions on atomic locations are duplicable?

We will return to this subtlety in §6.1; first, we will explain at a high level how protocols and exchanges are used in the steps of the algorithm, glossing over the details of ownership for atomics. Figure 6 shows the message passing and ownership transfer involved in the release-acquire pairs in terms of the predicates and exchanges used in the proof. Figure 7 contains the formal definitions. Note that these definitions refer to various pieces of ghost state (*e.g.,* tokens) projected from Γ. This Γ is just a record collecting together all the state associated with a particular instance of an RCU linked list.

Step 1 (Fig. 6a): After initializing a new node with value v at physical location x, the writer will own $x + \texttt{data} \hookrightarrow v$, as well as the $x + \texttt{link}$ field. Since in our RCU implementation it is possible that the physical x is a recycled node (*i.e.,* one that participated in the data structure previously but was deallocated and reallocated), we also create a fresh *abstract node ID*, i, which serves as a logical proxy for the *current* allocation of x. (Each time x is re-allocated, it will thus be associated with a different abstract ID. This simplifies protocol reasoning, enabling us, for instance, to use one-time token permissions to pass ownership of x back and forth between writer and readers, even though the physical x may in fact be passed back and forth multiple times if it is recycled.) This ID i will designate a token that a reader will have to use to get access to the node. The writer then splits the ownership of $x + \texttt{data}$ and $x + \texttt{link}$ into fractional pieces for the readers. For each reader thread $tid < N$, it transfers the fractional pieces designated for reader tid into an exchange, $\mathsf{PermX}(\Gamma, x, tid, i)$. The other side of this exchange is the (tid, i) token from the $\Gamma.\mathsf{rxtok}$ ghost state. Reader tid begins by owning all of the $\{tid\} \times \mathbb{N}$ of these tokens. (Compare this with the LX exchange created in Step 1 of the example in §5.2.)

Step 2 (Fig. 6b): We set up a protocol **LLP** on the `link` field, which formalizes Release-Acquire Pair 1 from §3. When the writer updates the `link` field of the parent p to point to x, it will store the knowledge that each $\mathsf{PermX}(\Gamma, x, tid, i)$ exchange has been created. (Compare this with **LFP** in Step 2 in §5.2.)

Step 3 (Fig. 6c): When reader tid reads $p + \texttt{link}$, it will learn about $\mathsf{PermX}(\Gamma, x, tid, i)$, and use its $\Gamma.\mathsf{rxtok}$ token to get access to x's fields.

Step 4 (Fig. 6d): When the reader is done with the node, it uses PermX in the opposite direction to get its $\Gamma.\mathsf{rxtok}$ token back. (Compare this and the previous step with Steps 3 and 4 in §5.2.)

Step 5 (Fig. 6e): Now, suppose later on the writer has deleted the node at location x, where x is associated with abstract node ID i, and now the writer wants to deallocate it. To do so, it increments `wcounter`, which in turn is governed by protocol **WCP** (formalizing Release-Acquire Pair 2 from §3). It stores $\mathsf{SnapshotValid}(H)$, which asserts that the abstract IDs for the nodes in the list are in fact in the state suggested by the "history snapshot" H. We explain history snapshots in §6.1, but intuitively, they represent the history of updates to the list, and H here is the most up-to-date history. In particular, since x has been deleted from the list, H here will mark x as a dead node. When reader tid sees the updated writer's counter, it infers that the writer wants to deallocate x because H marks x as dead.

Step 6 (Fig. 6f): For each of the abstract nodes i that the writer has requested for deallocation, the reader creates a $\mathsf{ModX}(\Gamma, tid, i)$ exchange, into which it transfers its (tid, i)-th $\Gamma.\mathsf{rxtok}$. The other side of the exchange is a corresponding (tid, i) token from $\Gamma.\mathsf{wxtok}$. Here, $\Gamma.\mathsf{wxtok}$ is a set of tokens that the writer starts with, which it uses to retrieve the reader's tokens. (Compare with the creation of the RX exchange in Step 5 in §5.2.)

Step 7 (Fig. 6f): The reader transmits its knowledge of the existence of these ModX exchanges by updating its counter, $\texttt{rcounters} + tid$. This counter is in turn governed by protocol $\mathbf{RCP}(\Gamma, tid)$,

$$\mathsf{PermX}(\Gamma, l, tid, i) : \Big((\exists v.\, l + \texttt{data} \overset{tid}{\hookrightarrow} v * P(v)) * (\exists j, L.\, \boxed{l + \texttt{link} : (i \cdot L, -) \mid \mathbf{LLP}(\Gamma, l, j)} * \boxed{j : \mathrm{Master}_{tid}(i \cdot L)}\Big)$$
$$\rightsquigarrow \boxed{\Gamma.\mathsf{rxtok} : \{(tid, i)\}}$$

$$\mathsf{ModX}(\Gamma, tid, i) : \boxed{\Gamma.\mathsf{rxtok} : \{(tid, i)\}} \rightsquigarrow \boxed{\Gamma.\mathsf{wxtok} : \{(tid, i)\}}$$

$$\mathsf{SnapshotValid}(\Gamma, H) \triangleq H \neq \texttt{nil} * \boxed{\Gamma.\mathsf{history} : \mathrm{Snapshot}(H)} * \mathsf{base}(H) \notin \mathsf{dead}(H) * H_*(\Gamma.q) = \mathsf{base}(H) \cdot \texttt{nil}$$
$$* \ (\forall l \in \mathrm{dom}(H_*).\, \exists j.\, \boxed{j : \mathrm{Snapshot}(H_*(l))} * \boxed{l + \texttt{link} : - \mid \mathbf{LLP}(\Gamma, l, j)}$$
$$* \ H^*(\mathsf{hd}(H_*(l))) \neq \top \Rightarrow \boxed{l + \texttt{link} : (H_*(l), H^*(\mathsf{hd}(H_*(l)))) \mid \mathbf{LLP}(\Gamma, l, j)})$$

Protocol	**State**	**Interpretation** (where x is the value stored at the location in question)	**Ordering** (State $\leq$ State$'$)
$\mathbf{LLP}(\Gamma, l, j)$	$(i_0 \cdot L_0,$ $i_1 \cdot L_1)$	$(i_1 \neq \mathsf{null} \Rightarrow x \neq 0 * H_*(x) = i_1 \cdot - * \forall t < N.\, \mathsf{exch}(\mathsf{PermX}(\Gamma, x, t, i_1)))$ $* \ (i_1 = \mathsf{null} \Rightarrow x = 0) * \exists H.\, \boxed{\Gamma.\mathsf{history} : \mathrm{Snapshot}(H)}$ $* \boxed{j : \mathrm{Snapshot}(i_0 \cdot L_0)} * H^*(i_0) = i_1 \cdot L_1 * H_*(l) = i_0 \cdot L_0$	Lexicographic, with $L_0 \leq L_0'$ if L_0 is a suffix of L_0'
$\mathbf{WCP}(\Gamma)$	(H, v)	$x = v * \boxed{\Gamma.\mathsf{ctok} : \{(N, v)\}} * \mathsf{SnapshotValid}(\Gamma, H)$	$(H = H' \wedge v \leq v')$ $\vee \ (H \leq H' \wedge v < v')$
$\mathbf{RCP}(\Gamma, tid)$	(H, v)	$x = v * \boxed{\Gamma.\mathsf{ctok} : \{(tid, v)\}} * (\forall i \in \mathsf{dead}(H).\, \mathsf{exch}(\mathsf{ModX}(tid, i)))$ $* \boxed{\Gamma.q + \texttt{wcounter} : (H, v) \mid \mathbf{WCP}(\Gamma)}$	Same as $\mathbf{WCP}(\Gamma)$

Figure 7. Exchanges and protocols for RCU.

formalizing Release-Acquire Pair 3 from §3. (Compare with the use of protocol **RFP** in Step 6 in §5.2.)

Step 8 (Fig. 6g): As the writer sees the updated $\texttt{rcounters} + tid$ fields, it uses each $\mathsf{ModX}(tid, i)$ it learns about to exchange its own (tid, i)-th $\Gamma.\mathsf{wxtok}$ token for the corresponding (tid, i)-th $\Gamma.\mathsf{rxtok}$ token. (Compare with the token exchange that occurs in Step 7 in §5.2.)

Step 9 (Fig. 6g): Finally, it uses these $\Gamma.\mathsf{rxtok}$ tokens with the $\mathsf{PermX}(\Gamma, x, tid, i)$ exchange to get back all the fractional permissions for x. After it has done this for for every reader's token, it will have collected the full permission for x, and it may deallocate the node. (Compare with the final Step 8 in §5.2.)

In the remainder of this section, we first present some more detail about the ghost state constructions needed in the proof, including those needed to account for ownership transfer of atomics (§6.1). We then explain the definitions of the abstract predicates ReaderSafe and SafePtr (from the spec in Figure 3) and sketch why the reader specifications are correct (§6.2). We conclude with a brief discussion of the extensions to our basic RCU implementation that are supported by our full verification (§6.3). The definition of WriterSafe and the full Hoare-style proofs are given in the appendix.

6.1 Ghost State

Our proof uses ghost state in three ways: (1) as permission tokens for exchanges, (2) to control the progress of protocols, and (3) to track the state (and more generally the history) of the linked list.

Exchange tokens We have the $\Gamma.\mathsf{rxtok}$ and $\Gamma.\mathsf{wxtok}$ tokens for the PermX and ModX exchanges. As we want a fresh token for each thread and for each abstract node ID, we take the PCM to be the powerset of $\{0, \ldots, N-1\} \times \mathbb{N}$ with disjoint union as composition. Reader tid starts with the set $\{tid\} \times \mathbb{N}$ of $\Gamma.\mathsf{rxtok}$. Meanwhile, the writer starts with all of the $\Gamma.\mathsf{wxtok}$.

Protocol state tokens Each thread has a counter which it alone is allowed to modify. The way we enforce this is by giving the thread a set of tokens, one for each state in the protocol associated with the counter. The interpretation function for the protocol then requires that to move to state s, the thread must give up the token which matches s. The thread begins with all of these tokens and deposits one each time it updates its counter. It knows that no other thread could have concurrently updated its counter, because it owns the unique token needed for the update.

For RCU, we use the powerset of $\{0, \ldots, N\} \times \mathbb{N}$ PCM for these tokens. The instance of this PCM is called $\Gamma.\mathsf{ctok}$ (for "counter token"). Reader tid starts with $\{tid\} \times \mathbb{N}$, and the writer starts with $\{N\} \times \mathbb{N}$. Then, we set up the interpretations of the **WCP** and **RCP** protocols for the counters so that each counter can only be updated to value v by the thread tid holding the appropriate counter token.

Master/snapshot PCM We will use a particular PCM construction to track the history of various objects in the RCU proof. The construction is a variant of the *authoritative monoid* described in Jung et al. [9]. This PCM allows a thread to update the history of an object by extending a non-duplicable "master" view of it. The PCM will also contain duplicable elements called "snapshots", which are partial, possibly stale, histories of the object. Readers use knowledge of these snapshots to establish lower bounds on the object's state.

Suppose P is a poset that represents a state transition system for some object. Suppose further that P has a least element, as well as an additional ordering property that for all $x, y, z \in P$, if $x \leq z$ and $y \leq z$, then either $x \leq y$ or $y \leq x$. We can then define a PCM whose elements have the form $\mathrm{Master}_k(p)$ and $\mathrm{Snapshot}(p)$, where $p \in P$ and k is a partial permission. Composition for this PCM is defined as follows (if the r.h.s. is undefined, so is the composition):

$$\mathrm{Master}_k(p) \cdot \mathrm{Master}_{k'}(p') \triangleq \mathrm{Master}_{k \oplus k'}(p), \text{ if } p = p'$$
$$\mathrm{Snapshot}(p) \cdot \mathrm{Master}_k(p') \triangleq \mathrm{Master}_k(p'), \text{ if } p \leq p'$$
$$\mathrm{Master}_k(p) \cdot \mathrm{Snapshot}(p') \triangleq \mathrm{Master}_k(p), \text{ if } p' \leq p$$
$$\mathrm{Snapshot}(p) \cdot \mathrm{Snapshot}(p') \triangleq \mathrm{Snapshot}(\max(p, p'))$$

We will write $\mathrm{Master}(p)$ as an abbreviation for $\mathrm{Master}_\top(p)$ (the full master permission).

To see why this construction is useful, imagine the RCU writer owns $\mathrm{Master}(p)$. This enables the writer to do two things. First, owning $\mathrm{Master}(p)$ entitles the writer to update it to any $\mathrm{Master}(p')$ such that $p' \geq p$. Formally, this is justified by GPS's "frame-preserving ghost update rule", which says that the update is valid so long as any PCM elements compatible with $\mathrm{Master}(p)$ are also compatible with

Master(p'). This "frame-preserving" condition guarantees that the writer's update does not invalidate the knowledge of other threads, and it holds here because indeed the only snapshots Snapshot(p'') compatible with Master(p) must have $p'' \leq p \leq p'$. Second, since Master(p) = Master(p) $\cdot$ Snapshot(p), owning Master(p) entitles the writer to fork off as many copies of Snapshot(p) as needed and transmit knowledge of them to readers through protocols. If a reader learns of Snapshot(p) through such a protocol, it then knows that p is a lower bound on the state of the object, *i.e.,* that the master copy must be in a state $p' \geq p$, and that if it ever learns of some other Snapshot(p'), it must be that either $p \leq p'$ or $p' \leq p$.

We will instantiate this definition with two different posets in the proof: the poset of *action histories*, which track the sequence of actions taken by the writer, and the poset of *abstract node ID histories*, which track the connection between a physical location and its logical proxies.

Action histories As part of Release-Acquire Pair 2, the writer needs to inform the readers that the node it wants to deallocate is no longer reachable. To do this, we record the history of the list as a piece of ghost state H, which is a list of abstract *actions* taken by the writer. Actions are of the form $\texttt{alloc}(l, i, i')$, $\texttt{upd}(i, i')$, or $\texttt{del}(i)$, where l is a location, $i \in \mathbb{N}$ and $i' \in \mathbb{N} \cup \{\mathsf{null}\}$. An $\texttt{alloc}(l, i, i')$ action represents allocating l and associating it with abstract node i, whose `link` field points to i' (which could be null). The $\texttt{upd}(i, i')$ action represents updating the `link` field of node i to point to i'. Finally, $\texttt{del}(i)$ indicates the writer's intention to deallocate node i.

Given a history H and an abstract location i, we can consider the subhistory of H containing only actions of the form $\texttt{alloc}(-, i, -)$, $\texttt{upd}(i, -)$, or $\texttt{del}(i)$. We call this the subhistory of H restricted to i, written H_i. For convenience, we treat H as a partial function, writing $H^*(i) = \top$ if $\texttt{del}(i) \in H$, and $H^*(i) = i_n \cdot \ldots \cdot i_1$ if $H_i = \texttt{alloc}(-, i, i_1) \cdot \texttt{upd}(i, i_2) \cdots \texttt{upd}(i, i_n)$. We can also consider the subtrace H_l of H containing only actions involving a physical location l. We define a similar partial function H_*, where $H_*(l) = i_n \cdot \ldots \cdot i_1$ if $H_l = \texttt{alloc}(l, i_1, -) \cdots \texttt{alloc}(l, i_n, -)$.

If the first action in H is $\texttt{alloc}(-, i, -)$, $\texttt{upd}(i, -)$, or $\texttt{del}(i)$, we say that the base of H, written $\mathsf{base}(H)$, is i. The $\mathsf{base}(H)$ is the abstract location corresponding to $q + \texttt{link}$, the pointer to the head of the list. We define $\mathsf{dead}(H)$ to be the set of all i such that $H(i) = \top$ and $\mathsf{live}(H) \triangleq \mathrm{dom}(H) \setminus \mathsf{dead}(H)$. We restrict the set of histories to "well-formed" ones, in which no node ever points to a dead node or an uninitialized node.

Histories can be ordered by saying that $H_1 \leq H_2$ if H_1 is a prefix of H_2. This ordering satisfies the following monotonicity properties, which we will use later in §6.2. If $H_1 \leq H_2$, then:

1. $\mathsf{dead}(H_1) \subseteq \mathsf{dead}(H_2)$, $\mathrm{dom}(H_1^*) \subseteq \mathrm{dom}(H_2^*)$, and $\mathrm{dom}((H_1)_*) \subseteq \mathrm{dom}((H_2)_*)$.
2. If $l \in \mathrm{dom}(H_1)$, then $(H_1)_*(l) \leq (H_2)_*(l)$, and if $i \in \mathsf{live}(H_1) \cap \mathsf{live}(H_2)$, then $H_1^*(i) \leq H_2^*(i)$.

It also satisfies the specific ordering property needed to use the master/snapshot PCM, *i.e.,* that $H_1 \leq H_3$ and $H_2 \leq H_3$ imply $H_1 \leq H_2$ or $H_2 \leq H_1$. In our RCU proof, $\Gamma.\mathsf{history}$ is an instance of this snapshot PCM. The writer is the thread that owns the authoritative Master(H), putting it in a privileged position. First of all, the thread is allowed to update the state of the history PCM to Master(H') so long as $H' \geq H$, *i.e.,* so long as the writer only *extends* the history with new actions that do not invalidate any existing snapshots. Second, the writer can copy off duplicable snapshots of this master, and then store them in the $q + \texttt{wcounter}$ counter and the `link` fields of the nodes. Since these snapshots are duplicable, they can be passed to readers as part of the protocol for Release-Acquire Pair 2. If a reader has $\boxed{\text{Snapshot}(H) : \mathsf{history}}$, then it knows that this snapshot H is a lower bound on the state of the master history. Consequently, once the reader learns that an abstract node i is in $\mathsf{dead}(H)$, it knows that the master copy can never revive i without violating dead set monotonicity, so it is safe for the reader to give up its rxtok tokens for i in Step 6 of the proof. Once an abstract node is dead, it stays dead.

"Atomic ownership" and abstract node ID histories In our explanation of the RCU proof, we described the $\mathsf{PermX}(\Gamma, l, tid, i)$ exchange as a way to transfer fractional ownership of a physical node l (with abstract ID i) back and forth between the writer and the tid-th reader. For the nonatomic `data` component of a node l, it is clear what this means: PermX serves to transfer the fractional permission $l + \texttt{data} \overset{tid}{\longhookrightarrow} v$ (along with the knowledge of the per-item invariant $P(v)$) back and forth. However, as noted at the beginning of §6, it is not clear what it means to transfer (fractional) ownership of l's *atomic* component—namely, its `link` field. Atomic locations are not (fractionally) ownable: they are governed by shared protocols and may be read/written concurrently.

Indeed, when the writer transfers "ownership" of l to reader tid, it does not actually transfer ownership of its atomic $l + \texttt{link}$ field, as this is not possible. Rather, the writer uses PermX to transfer several things which collectively suffice to enable the reader to safely access the $l + \texttt{link}$ field. First, it transfers the knowledge that $l + \texttt{link}$ obeys the **LLP** protocol (see Step 2 of the proof outline above, and more below). Second, it transfers the knowledge that the physical location l is *currently* associated with abstract ID i and that l will not be recycled and reassociated with any other abstract ID until the reader gives back its fractional ownership of the node to the writer. This is important because the reader will depend on i being a logical proxy for l in the proof; if l were to be recycled prematurely, any reasoning that the reader did based on i would not be sound.

Formally, we encode the knowledge about the association between physical locations l and abstract IDs i—along with the permission to reassociate locations with new abstract IDs when they are recycled—as elements of a second master/snapshot PCM. For each l, we keep a list of which abstract node IDs it has been associated with. The head of the list represents the node's current abstract ID. We can impose a partial ordering on these lists by saying that $L \leq L'$ if L is a suffix of L'. (This ordering satisfies the additional property needed to use the master/snapshot construction.)

When the writer first allocates a node at physical location l, it associates l with a fresh abstract node ID i by creating a new master instance j of the above PCM, initialized to store the singleton list $[i]$. Via the PermX exchange, the writer then transfers fractional ownership of j to each of its readers (*i.e.,* it gives reader tid $\boxed{j : \text{Master}_{tid}([i])}$). With this fractional ownership in hand, the readers know that the writer cannot possibly reassociate l with a different abstract ID (by advancing the state of j) because that would require it to own the full master copy of j.

Finally, the writer assigns $l + \texttt{link}$ the protocol $\mathbf{LLP}(\Gamma, l, j)$. The states of this protocol are pairs of nonempty lists $(i_0 \cdot L_0, i_1 \cdot L_1)$, where the first component represents the list of abstract IDs that l has been associated with (i_0 is the current one), and the second represents the sequence of nodes its `link` field has pointed to during the period of l's association with i_0. Note that the **LLP** protocol's interpretation of this state includes a snapshot of the PCM instance j, with state $i_0 \cdot L_0$. When the readers read $l + \texttt{link}$, this snapshot, together with their fractional ownership of j, lets them conclude that i_0 is the same abstract ID i that they know about from PermX.

During synchronization, the writer will collect all the fractional pieces of j back, so that it can safely deallocate l. The writer maintains the invariant that it owns the full master j for every node in the deallocated node pool. Hence, if the writer ends up recycling l to represent a new node, it will be able to associate a fresh abstract node ID i' with l by pushing i' onto the head of the list stored in j.

6.2 Reader Abstract Predicates

We now give the parts of the definitions of ReaderSafe and SafePtr that are relevant for accessing nodes in the list:

$$\begin{aligned}&\mathsf{ReaderSafe}(q, H, tid) \triangleq \exists \Gamma.\ \Gamma.q = q * \mathsf{SnapshotValid}(\Gamma, H)\\ &\quad * \boxed{\Gamma.\mathsf{rxtok} : \{tid\} \times (\mathbb{N} \setminus \mathsf{dead}(H))} * \text{(counter resources)}\\ &\quad * \exists j.\ \boxed{q + \texttt{link} : (\mathsf{base}(H), H^*(\mathsf{base}(H)))} \boxed{\mathbf{LLP}(\Gamma, q, j)}\\ &\qquad * \boxed{j : \mathrm{Master}_{tid}(\mathsf{base}(H))}\\ &\mathsf{SafePtr}(q, H, p) \triangleq \exists \Gamma.\ \Gamma.q = q * \mathsf{SnapshotValid}(\Gamma, H)\\ &\quad * (p \neq 0 \Rightarrow \exists i.\ i \notin \mathsf{dead}(H)\\ &\qquad * \forall tid < N.\ \mathsf{exch}(\mathsf{PermX}(\Gamma, p, tid, i)))\end{aligned}$$

The $\mathsf{ReaderSafe}(q, H_1, tid)$ predicate asserts that H_1 is a valid snapshot and contains all the $\Gamma.\mathsf{rxtok}$ tokens for thread tid except for the nodes that are dead in H_1. In addition, the reader is given partial "ownership" of $q + \texttt{link}$ (whose abstract node ID is fixed to be $\mathsf{base}(H_1)$ since $q + \texttt{link}$ is never deallocated). From this definition, it is clear how we lose $\mathsf{ReaderSafe}(q, H_1, tid)$ during `rcuQuiescentState`, and get back $\mathsf{ReaderSafe}(q, H_2, tid)$ for some H_2. During this function, the reader will transfer some of its $\Gamma.\mathsf{rxtok}$ into exchanges. However, it learns that $\mathsf{SnapshotValid}(H_2)$ for some new H_2, and only gives up tokens in $\mathsf{dead}(H_2)$.

$\mathsf{SafePtr}(q, H_1, p)$ just says that if p is non-null, then there exists a $\mathsf{PermX}(\Gamma, p, tid, i)$ exchange for some $i \notin \mathsf{dead}(H_1)$. By the definition of $\mathsf{ReaderSafe}(q, H_1, tid)$, the reader thus knows that it must have $\boxed{\Gamma.\mathsf{rxtok} : (tid, i)}$, so it can use this, together with the PermX exchange, to gain access to the node located at p. From this we can see why the precondition for `rcuReadNext` is sufficient.

The postcondition for `rcuReadNext` requires us to prove that when the reader does an acquire read on $p + \texttt{link}$ (line 9) and gets some value p', that $\mathsf{SafePtr}(q, H_1, p')$ is also true. Now, we know from $\mathsf{PermX}(\Gamma, p, tid, i)$ that p is associated with abstract node ID i, since the exchange contains a piece of the master PCM element listing all the abstract IDs that p has been associated with. This list has the form $i \cdot L$ for some L. Since this is part of the master copy, no other thread could have associated p with some new ID. When we read $p + \texttt{link}$, the protocol guarantees that if $p' \neq 0$, then the protocol state is of the form $(i \cdot L, i' \cdot L')$, and there exists some snapshot of an H_2 such that $H_2^*(i) = i' \cdot L'$ and $(H_2)_*(p) = i \cdot L$. In addition, we also learn that there are $\mathsf{PermX}(\Gamma, p', tid, i')$ for each tid. This gives us most of what we need in order to establish $\mathsf{SafePtr}(q, H_1, p')$—it merely remains to show that $i' \notin \mathsf{dead}(H_1)$.

Now, because of the rules for snapshot composition, either $H_1 \leq H_2$ or $H_2 \leq H_1$. In the former case, we are done, because $i' \notin \mathsf{dead}(H_2)$, and as noted in §6.1, dead sets grow monotonically. In the latter case, where $H_2 \leq H_1$, the monotonicity properties mentioned in §6.1 give us that that $(H_1)_*(p) \geq (H_2)_*(p) = i \cdot L$. However, by SnapshotValid, the reader has a snapshot of the abstract IDs that matches $(H_1)_*(p)$. Since this snapshot cannot be bigger than the master, we have $(H_1)_*(p) \leq i \cdot L$. Hence, we must have that $(H_1)_*(p) = i \cdot L$. In addition, from $\mathsf{SafePtr}(q, H_1, p)$, we know that $i \notin \mathsf{dead}(H_1)$ and thus that $H_1^*(i) \neq \top$. Using SnapshotValid again, this means that the reader already had a protocol assertion about $p + \texttt{link}$ that said it was at least in state $(i \cdot L, H_1^*(i))$. Therefore, we must have $(i \cdot L, H_1^*(i)) \leq (i \cdot L, i' \cdot L')$, which implies $H_1^*(i) \leq i' \cdot L'$. Using the monotonicity properties from §6.1 once more, since $H_1 \geq H_2$, we have that $H_1^*(i) \geq H_2^*(i) = i' \cdot L'$. Hence, $H_1^*(i) = i' \cdot L'$, and so $i' \notin \mathsf{dead}(H_1)$.

6.3 Extensions to the Basic RCU Implementation

The full version of RCU verified in the appendix contains two additional features:

- the writer batches together several node deallocations, and
- readers dynamically register themselves.

Supporting batched deallocation is straightforward. In the call to `rcuNodeUpdate`, the writer adds the old node to a deallocation stack. Then, when it wants to deallocate the stack, it performs `rcuSynchronize`, and gets the reader tokens for *all* nodes in the stack at once.

For dynamic registration, the RCU metadata contains an additional field, $q + \texttt{numreaders}$, which is a counter storing the number of readers. To register, a reader does a fetch-and-increment on this field to get their tid. During `rcuSynchronize`, the writer reads $q + \texttt{numreaders}$ and only examines the $q + \texttt{rcounters}$ entries for registered readers. For the proof, we have a protocol on $q + \texttt{numreaders}$ that initially contains $\mathsf{ReaderSafe}(q, H, tid)$ for all tid. During the fetch-and-increment, the reader takes out the $\mathsf{ReaderSafe}(q, H, tid)$ for the tid it gets assigned. Similarly, during `rcuSynchronize`, when the writer wants to deallocate the node currently associated with logical ID i, it takes out $\Gamma.\mathsf{rxtok}(tid, i)$ for all tids that have not yet been assigned to registered readers.

7. Related Work

Consume reads. The C11 memory model also includes *consume* reads, which are weaker than acquire reads and cheaper to implement efficiently on the Power and ARM architectures. With acquire reads, *everything* following the read is guaranteed to happen after the things preceding the matching release write. However, with consume reads, only the things that have a data dependency on the value read are guaranteed to happen after. For example, consider:

$$\begin{array}{l||l}
[x]_{\texttt{na}} := 25; & \texttt{let } p = [m]_{\texttt{cons}} \\
[y]_{\texttt{na}} := 37; & \texttt{if } (p \texttt{ != } 0)\ \{ \\
[m]_{\texttt{rel}} := x; & \quad [a]_{\texttt{na}} := [p]_{\texttt{na}}; \\
 & \quad [b]_{\texttt{na}} := [y]_{\texttt{na}}; \\
 & \}
\end{array}$$

In this example, the right thread's read through the pointer p is guaranteed to happen after the write that initialized x, because this access depends on the value from the consume read of m. In contrast, the access to y is racy, because it does not have a data dependency, only a control dependency. This ordering is sufficient for RCU if the reader does consume reads on the `link` field, because the accesses to the fields of the node will depend on the pointer it reads. In fact, supporting RCU was a primary motivation for including consume reads in the C11 standard. However, the standard may be revised because the current rules for data dependency tracking are too complicated, and most compilers treat consume reads as acquire reads [14]. Once these revisions are finalized, we believe it should be possible to extend GPS with support for consume reads through a modality that tracks dependencies.

User-space RCU. Our implementation of RCU is based on that of Desnoyers et al. [5], who describe a number of RCU implementations, of which QSBR is one that provides highly optimized performance. Our implementation differs from theirs in a few ways. First, their implementation uses memory barriers rather than C11 concurrency primitives. At present, there are no program logics for C11 that are as rich as GPS (in its support for protocols) and that also handle release/acquire fences and relaxed accesses. However, we believe that, assuming some handling of such mechanisms is developed in the way we imagine should be possible, our message-passing explanation will still suffice, without requiring us to revert to the notion of a grace period (see below). In particular, if the appropriate logic existed, we believe the following relaxations would be possible without changing the basic structure of our proof:

- All the release writes in `rcuNew` (lines 2, 3, 4), the write at line 19 in `rcuNodeUpdate`, and the write at line 13 in `rcuNodeAppend`

could be made relaxed (or even non-atomic). In fact, these are initialization writes, so no explicit release fence is needed.
- The reads at lines 7 and 9 could be made consume reads, as explained above.
- The read at line 32 could be made relaxed, provided we add an acquire fence after line 33.
- The reads at lines 16, 24, and 28 could be made relaxed (or non-atomic) because only the same thread can write to the field. Doing so should not affect the proof, as no real ownership transfer is performed.

In addition, Desnoyers et al.'s implementation of QSBR allows readers to go "offline" for extended periods by setting their counter field to 0. The writer's counter starts at 1, and when the writer performs `rcuSynchronize`, it checks that each reader's counter either matches its own or is 0. Later, the reader can go back online by copying the writer's counter value again. We left a proper treatment of this extension for future work because it requires a combination of weak-memory primitives and stronger synchronization operations (SC fences), for which no adequate verification techniques presently exist.

Other RCU verifications. Gotsman et al. [7] verify an RCU-based non-blocking stack implementation under a sequentially consistent memory model. Their RCU synchronization procedure is closer to exclusively using the offline/online feature of QSBR described above. They formalize the concept of a *grace period*, which is often used to informally explain RCU. A grace period is the length of time from when a node becomes unreachable until no readers are accessing it any longer. They show that this concept can be used to structure the proofs of related memory management techniques such as hazard pointers and epoch-based reclamation. Their proof uses a concurrent separation logic extended with temporal operators to make statements about the grace period. It would be interesting to try to add such temporal operators to a logic like GPS and see if a proof based on grace periods can be formalized in the setting of weak memory, but as we have shown, one can verify an RCU implementation even without them.

Alglave et al. [2] use a bounded model checker to examine a real implementation of RCU taken from the Linux kernel, which uses explicit hardware fences rather than the new C11 concurrency primitives. They apply their tool to a test harness running one reader and one writer concurrently, and verify (on several architectures) that the reader will not see malformed or uninitialized data. In contrast, we consider a simpler implementation of RCU, but provide a general proof of correctness against a modular Hoare-style specification.

Relaxed Separation Logic (RSL). One reason perhaps why there has been no prior work on formally verifying RCU in a weak-memory setting is that program logics for weak-memory concurrency have only begun appearing very recently. For instance, it was only in 2013 that Vafeiadis and Narayan [19] proposed RSL, the first program logic for the C11 memory model. RSL is a simpler logic than its sequel, GPS, and also less powerful: the GPS paper presents several examples that are beyond the scope of RSL. It is therefore instructive to consider whether we could have verified our RCU implementation using the simpler RSL. We cannot provide a definite answer whether this is possible or not, but we believe it is rather unlikely, given how heavily our proof relies on the rely-guarantee reasoning afforded by protocols, which is not directly supported in RSL.

Acknowledgments

This research was supported in part by an NDSEG fellowship from the US Department of Defense, by the Air Force Office of Scientific Research under Award No. FA9550-12-1-0370, by the EC FP7 FET project ADVENT, and by an internship from MPI-SWS. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the supporting organizations.

References

[1] Supplemental material for this paper available at the following URL: http://plv.mpi-sws.org/gps/rcu/.

[2] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, 2013.

[3] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[4] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, 2005.

[5] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *IEEE Trans. Parallel Distrib. Syst.*, 23(2):375–382, 2012.

[6] E. W. Dijkstra. EWD123: Cooperating Sequential Processes. Technical report, 1965.

[7] A. Gotsman, N. Rinetzky, and H. Yang. Verifying concurrent memory reclamation algorithms with grace. In *ESOP*, 2013.

[8] ISO/IEC 9899:2011. Programming language C.

[9] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, 2015.

[10] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9): 690–691, 1970.

[11] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In *CAV*. 2012.

[12] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[13] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *PDCS*, 1998.

[14] P. E. McKenney, T. Riegel, J. Preshing, H. Boehm, C. Nelson, and O. Giroux. N4215: Towards implementation and use of memory_order_consume, 2014. Available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4215.pdf.

[15] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[16] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *CACM*, 53(7):89–97, 2010.

[17] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, 2014.

[18] V. Vafeiadis. Concurrent separation logic and operational semantics. In *MFPS*, volume 276 of *ENTCS*, 2011.

[19] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*, 2013.

[20] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL*, 2015.