Concurrent Data Structures Linked in Time

Germán Andrés Delbianco[1,2], **Ilya Sergey**[3], **Aleksandar Nanevski**[1], **and Anindya Banerjee**[1]

1 **IMDEA Software Institute, Madrid, Spain**
{german.delbianco, aleks.nanevski,anindya.banerjee}@imdea.org
2 **Universidad Politécnica de Madrid, Spain**
3 **University College London, UK**
i.sergey@ucl.ac.uk

Abstract

Arguments about correctness of a concurrent data structure are typically carried out by using the notion of *linearizability* and specifying the linearization points of the data structure's procedures. Such arguments are often cumbersome as the linearization points' position in time can be *dynamic* (depend on the interference, run-time values and events from the past, or even future), *non-local* (appear in procedures other than the one considered), and whose position in the execution trace may only be determined after the considered procedure has already terminated.

In this paper we propose a new method, based on a separation-style logic, for reasoning about concurrent objects with such linearization points. We embrace the dynamic nature of linearization points, and encode it as part of the data structure's *auxiliary state*, so that it can be dynamically modified in place by auxiliary code, as needed when some appropriate run-time event occurs. We name the idea *linking-in-time*, because it reduces temporal reasoning to spatial reasoning. For example, modifying a temporal position of a linearization point can be modeled similarly to a pointer update in separation logic. Furthermore, the auxiliary state provides a convenient way to concisely express the properties essential for reasoning about clients of such concurrent objects. We illustrate the method by verifying (mechanically in Coq) an intricate optimal snapshot algorithm due to Jayanti, as well as some clients.

Keywords and phrases Concurrent separation logics, Linearization points, Snapshot Algorithms, FCSL

1 Introduction

Formal verification of concurrent objects commonly requires reasoning about linearizability [19]. This is a standard correctness criterion whereby a concurrent execution of an object's procedures is proved equivalent, via a simulation argument, to some sequential execution. The clients of the object can be verified under the sequentiality assumption, rather than by inlining the procedures and considering their interleavings. Linearizability is often established by describing the *linearization points* (LP) of the object, which are points in time where procedures take place, *logically*. In other words, even if the procedure physically executes across a time interval, exhibiting its linearization point enables one to pretend, for reasoning purposes, that it occurred instantaneously (*i.e.*, atomically); hence, an interleaved execution of a number of procedures can be reduced to a sequence of atomic events.

Reasoning about linearization points can be tricky. Many times, a linearization point of a procedure is not *local*, but may appear in another procedure or thread. Equally bad, linearization points' place in time may not be determined statically, but may vary based on the past, and even future, *run-time* information, thus complicating the simulation arguments. A particularly troublesome case is when run-time information influences the logical order of a procedure that has already terminated. This paper presents a novel approach to specification

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of concurrent objects, in which the dynamic and non-local aspects inherent to linearizability can be represented in a procedure-local and thread-local manner.

The starting point of our idea is to realize what are the shortcomings of linearizability as a canonical specification method for concurrent objects. Consider, for instance, the following two-threaded program manipulating a correct implementation of stack by invoking its `push` and `pop` methods, which are atomic, *i.e.*, linearizable:

```
push(3);        || push(4)
t1 := pop();    || t2 := pop();
```

Assuming that the execution started in an empty stack, we would like to derive that it returns an empty stack and `(t1, t2)` is either `(3, 4)` or `(4, 3)`. Linearizability of the stack guarantees that the overall trace of `push`/`pop` calls is coherent with respect to a sequential stack execution. However, it does not capture *client*-specific partial knowledge about the *ordering* of particular `push`/`pop` invocations in sub-threads, which is what allows one to prove the desired result as a composition of separately-derived partial specifications of the left and the right thread.

This thread-local information, necessary for compositional reasoning about clients, can be captured in a form of *auxiliary state* [33] (a generalization of *history variables* [2]), widely used in Hoare-style specifications of concurrent objects [23, 24, 27, 37]. A testament of expressivity of Hoare-style logics for concurrency with rich auxiliary state are the recent results in verification of fine-grained data structures with helping [37], concurrent graph manipulations [36], barriers [10, 23], and even *non-linearizable* concurrent objects [38].

Although designed to capture information about events that happened concurrently *in the past* (hence the original name *history variables*), auxiliary state is known to be of little use for reasoning about data structures with *speculative* executions, in which the ordering of past events may depend on other events happening in the *future*. Handling such data structures requires specialized metatheory [28] that does not provide convenient abstractions such as auxiliary state for client-side proofs. This is one reason why the most expressive client-oriented concurrency logics to date avoid reasoning about speculative data structures altogether [23].

Our contributions

The surprising result we present in this paper is that by allowing certain *internal* (*i.e.*, not observable by clients) manipulations with the auxiliary state, we can use an existing program logic for concurrency, like, *e.g.*, FCSL [31, 36], to specify and verify algorithms whose linearizability argument requires speculations, *i.e.*, depends on the *dynamic reordering* of events based on run-time information from the future. To showcase this idea, we provide a new specification (spec) and the first formal proof of a very sophisticated snapshot algorithm due to Jayanti [22], whose linearizability proof exhibits precisely such kind of dependence.

While we specify Jayanti's algorithm by means of a separation-style logic, the spec nevertheless achieves the same general goals as linearizability, combined with the benefits of compositional Hoare-style reasoning. In particular, our Hoare triple specs expose the logical atomicity of Jayanti's methods (Section 3), while hiding their true fine-grained and physically non-atomic nature. The approach also enables that the separation logic reasoning is naturally applied to clients (Section 4). Similarly to linearizability, our clients can reason out of procedures' spec, not code. We can also ascribe the same spec to different snapshot algorithms, without modifying client's code or proof.

```
write $(p, v)$ {
  $p := v$;
  $b \leftarrow \mathsf{read}(S)$;
  if $b$
  then $(\texttt{fwd}\ p) := v$}

fwd $(p : \texttt{ptr})$ {
  return $(p = x)$ ? $fx$: $fy$ }

scan : $(A \times A)$ {
   $S :=$ true;
   $fx := \bot$;
   $fy := \bot$;
   $vx \leftarrow \mathsf{read}(x)$;
   $vy \leftarrow \mathsf{read}(y)$;
   $S :=$ false;
   $ox \leftarrow \mathsf{read}(fx)$;
   $oy \leftarrow \mathsf{read}(fy)$;
   $rx \leftarrow$ if $(ox \neq \bot)$ then $ox$ else $vx$;
   $ry \leftarrow$ if $(oy \neq \bot)$ then $oy$ else $vy$;
   return $(rx, ry)$}
```

Figure 1 Jayanti's single-scanner/single-writer snapshot algorithm.

In more detail, our approach works as follows. We use shared auxiliary state to record, as a list of timed events (*e.g.*, writes occurring at a given time), the logical order in which the object's procedures are perceived to execute, each instantaneously (Section 5). Tracking this time-related information through state enables us to specify its dynamic aspects. We can use *auxiliary code* to mutate the logical order *in place*, thereby permuting the logical sequencing of the procedures, as may be needed when some run-time event occurs (Sections 6 and 7). This mutation is similar to updating pointers to reorder a linked list, except that it is executed over auxiliary state storing time-related data, rather than over real state. This is why we refer to the idea as *linking-in-time*.

Encoding temporal information by way of representing it as mutable state allows us to use FCSL off-the-shelf to verify example programs. In particular, FCSL has been implemented in the proof assistant Coq, and we have fully mechanized the proof of Jayanti's algorithm [1].

2 Verification challenge and main ideas

Jayanti's snapshot algorithm [22] provides the functionality of a shared array of size m, operated on by two procedures: `write`, which stores a given value into an element, and `scan`, which returns the array's contents. We use the *single-writer/single-scanner* version of the algorithm. which assumes that at most one thread writes into an element, and at most one thread invokes the scanner, at any given time. In other words, there is a scanner lock and m per-element locks. A thread that wants to scan, has to acquire the scanner lock first, and a thread that wants to write into element i has to acquire the i-th element lock. However, scanning and writing into different elements can proceed concurrently. This is the simplest of Jayanti's algorithms, but it already exhibits linearization points of dynamic nature. We also restrict the array size to $m = 2$ (*i.e.*, we consider two pointers x and y, instead of an array). This removes some tedium from verification, but exhibits the same conceptual challenges.

The difficulty in this snapshot algorithm is ensuring that the scanner returns the most recent snapshot of the memory. A naïve scanner, which simply reads x and y in succession, is unsound. To see why, consider the following scenario, starting with $x = 5$, $y = 0$. The scanner reads x, but before it reads y, another thread preempts it, and changes x to 2 and, subsequently, y to 1. The scanner continues to read y, and returns $x = 5, y = 1$, which was never the contents of the memory. Moreover, (x, y), changed from $(5, 0)$ to $(2, 0)$ to $(2, 1)$ as a result of distinct non-overlapping writes; thus, it is impossible to find a linearization point

```
l:  write (x,2);  ||              ||
    write (y,1)   ||  c:  scan () ||  r:  write (x,3)
```

(a) Parallel composition of three threads `l, c, r`.

```
1  c: S := true
2  c: fx := ⊥
3  c: fy := ⊥
4  c: read(x)  // vx <- 5
5  c: read(y)  // vy <- 0
6  l: x := 2
7  l: read(S)  // b <- true
8  l: fx := 2
9  l: return ()
10 r: x := 3
11 l: y := 1
12 l: read(S)  // b <- true
13 l: fy := 1
14 l: return ()
15 c: S := false
16 r: read(S)  // b <- false
17 r: return ()
18 c: read(fx) // ox <- 2
19 c: read(fy) // oy <- 1
20 c: return (2,1)
```

(b) A possible interleaving of the threads in (a).

Figure 2 An example leading to a scanner miss.

for the scan because linearizability only permits reordering of overlapping operations.

To ensure a sound snapshot, Jayanti's algorithm internally keeps additional *forwarding pointers fx* and *fy*, and a boolean *scanner bit S*. The implementation is given in Figure 1.[1] The intuition is as follows. A writer storing v into p (line 2), will additionally store v into the forwarding pointer for p (line 5), provided S is set. If the scanner missed the write and instead read the old value of p (lines 10–11), it will have a chance to catch v via the forwarding pointer (lines 13–14). The scanner bit S is used by writers (line 3) to detect a scan in progress, and forward v.

As Jayanti proves, this implementation *is* linearizable. Informally, every overlapping calls to `write` and `scan` can be rearranged to appear as if they occurred sequentially. To illustrate, consider the program in Figure 2a, and one possible interleaving of its primitive memory operations in Figure 2b. The threads `l`, `c`, and `r`, start with $x = 5, y = 0$. The thread `c` is scheduled first, and through lines 1–5 sets the scanner bit, clears the forwarding pointers, and reads $x = 5, y = 0$. Then `l` intervenes, and in lines 6–9, overwrites x with 2, and seeing S set, forwards 2 to *fx*. Next, `r` and `l` overlap, writing 3 into x and 1 into y. However, while 1 gets forwarded to *fy* (line 13), 3 is not forwarded to *fx*, because S was turned off in line 15 (*i.e.*, the scan is no longer in progress). Hence, when `c` reads the forwarded values (lines 18, 19), it returns $x = 2, y = 1$.

While $x = 2, y = 1$ was never the contents of the memory, returning this snapshot is nevertheless justified because we can *pretend* that the scanner *missed* `r`'s write of 3. Specifically, the events in Figure 2b can be *reordered* to represent the following sequential execution:

$$\texttt{write}\,(\texttt{x}, 2);\ \texttt{write}\,(\texttt{y}, 1);\ \texttt{scan}\,();\ \texttt{write}\,(\texttt{x}, 3) \tag{1}$$

Importantly, the client programs have no means to discover that a different scheduling actually took place in real time, because they can access the internal state of the algorithm only via interface methods, `write` and `scan`.

This kind of temporal reordering is the most characteristic aspect of linearizability proofs, which typically describe the reordering by listing the linearization points of each procedure.

[1] Following Jayanti, we simplify the presentation and omit the locking code that ensures the single-writer/single-scanner setup. Of course, in our Coq development [1], we make the locking explicit.

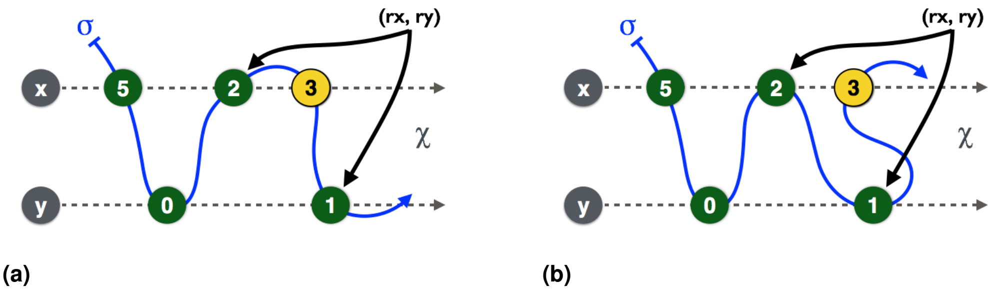

Figure 3 Changing the logical ordering (solid line σ) of write events from (5, 0, 2, 3, 1) in (a) to (5, 0, 2, 1, 3) in (b), to reconcile with `scan` returning the snapshot $x = 2, y = 1$, upon missing the write of 3. Dashed lines χ represent real-time ordering.

At a linearization point, the procedure's operations can be spliced into the execution history as an uninterrupted chunk. For example, in Jayanti's proof, the linearization point of `scan` is at line 12 in Figure 1, where the scanner bit is unset. The linearization point of `write`, however, may vary. If `write` starts before an overlapping `scan`'s line 12, and moreover, the `scan` misses the `write`—note the dynamic and future-dependent nature of this property—, then `write` should appear after `scan`; that is, the `write`'s linearization point is right after `scan`'s linearization point at line 12. Otherwise, `write`'s linearization point is at line 2. In the former case, `write` exactly has a non-local and future-dependent linearization point, because the decision on the logical order of this `write` depends on the execution of `scan` in a different thread. This decision takes effect on lines 13–14, which can take place *after* the execution of `write` has terminated. For instance, in Figure 2b the execution of `write` in `r` terminates at step 17, yet, in Jayanti's proof, the decision to linearize this `write` after the overlapping `scan` is taken at line 18, when the `scan` reads the value from the previous `write`.

Obviously, the high-level pattern of the proof requires tracking the *logical ordering* of the `write` and `scan` events, which differs from their *real-time ordering*. As the logical ordering is inherently dynamic, depending on properties such as `scan` missing a `write`, we formalize it in Hoare logic, by keeping it as a list of events in auxiliary state that can be dynamically reordered as needed. For example, Figure 3 shows the situation in the execution of `scan` that we reviewed above. We start with the (initializing) writes of 5 and 0 already executed, and our program performs the writes of 2, 3 and 1 in the real time order shown by the position of the events on the dashed lines. In Figure 3a, the logical order σ coincides with real-time order, but is unsound for the snapshot $x = 2, y = 1$ that `scan` wants to return. In that case, the auxiliary code with which we annotate `scan`, will change the sequence σ in-place, as shown in Figure 3b.

Our specification and verification challenge then lies in reconciling the following requirements. First, we have to posit specs that say that `write` performs a write, and `scan` performs a scan of the memory, with the operations executing in a single logical moment. Second, we need to implement the event reordering discipline so that a method call only reorders events that overlap with it; the logical order of the past events should be preserved. This will be accomplished by introducing yet further structures into the auxiliary state and code. Finally, the specs must hide the specifics of the reordering discipline, which should be internal to the snapshot object. Different snapshot implementations should be free to implement different reorderings, without changing the method specs.

3 Specification

General considerations. For the purposes of specification and proof, we record a history of the snapshot object as a set of entries of the form $t \mapsto (p, v)$. The entry says that at time t (a natural number), the value v was written into the pointer p. We thus identify a write event with a *single* moment in time t, enabling the specs of `write` and `scan` to present the view that write events are logically atomic. Moreover, in the case of snapshots, we can ignore the scan events in the histories. The latter do not modify the state in a way observable by clients who can access the shared pointers only via interface methods `write` and `scan`.

We keep three auxiliary history variables. The history variables χ_{s} and χ_{o} are local to the specified thread, and record the *terminated* write events carried out by the specified thread, and that thread's interfering environment, respectively. We refer to χ_{s} as the *self*-history, and to χ_{o} as the *other*-history [27,30,31,37]. The role of χ_{o} is to enable the spec of `write` to situate the performed write event within the larger context of past and ongoing writes, and the spec of `scan` to describe how it logically reordered the writes that overlapped with it. The third history variable χ_{j} records the set of write events that are in progress. These are events that have been initiated, timestamped, and have executed their physical write to memory, but have not terminated yet. It is an important component of our auxiliary state design that when a write event terminates, it is moved from χ_{j} to the invoking thread's χ_{s}, to indicate the *ownership* of the write by the invoking thread. We name by χ the union $\chi_{\mathsf{s}} \mathbin{\dot\cup} \chi_{\mathsf{o}} \mathbin{\dot\cup} \chi_{\mathsf{j}}$, which is the global history of the data structure. As common in separation logic, the union is *disjoint*, *i.e.*, it is undefined if the components contain duplicate timestamps. By the semantics of our specs, χ is always defined, thus χ_{s}, χ_{o} and χ_{j} never duplicate timestamps.

The real-time ordering of the timestamped events is the natural numbers ordering on the timestamps. To track the *logical* ordering, we need further auxiliary notions. The first is the auxiliary variable σ, whose type is a mathematical sequence. The sequence σ is a permutation of timestamps from χ showing the logical ordering of the events in χ. We write $t_1 \leqslant_\sigma t_2$, and say that t_1 is logically ordered before t_2, if t_1 appears before t_2 in σ. The sequence σ resides in joint state, and can be dynamically modified by any thread. For example, the execution of the scanner may reorder σ, as shown in Figure 3b. Because σ is a sequence, the order $\leqslant_\sigma$ is linear.

Because sequence σ changes dynamically under interference, it is not appropriate for specifications. Thus, our second auxiliary notion is the *partial* order Ω, a suborder of $\leqslant_\sigma$ that is *stable* in the following sense. It relates the timestamps of events whose logical order has been determined, *and will not change in the future*. Thus Ω can grow over time, to add new relations between previously unrelated timestamps, but cannot change the old relations.

To illustrate the distinction between the two orders, we refer to Figure 3a. There, σ represents the linear order 5—0—2—3—1, which changes in Figure 3b to 5—0—2—1—3. Since 1 and 3 exchange places, the stable order Ω cannot initially relate the two. Thus, in Figure 3a, Ω is represented by the Hasse diagram $5{-}0{-}2{<}\substack{1\\3}$. In Figure 3b, the relation 1—3 is added to this partial order, making it the linear order 5—0—2—1—3. Note how the previous relations remain unchanged.

The third auxiliary notion is the set $\mathsf{scanned}\,\Omega$ of timestamps. A write's timestamp is placed in $\mathsf{scanned}\,\Omega$, if that write has been observed by some scanner; that is, the written value is returned in some snapshot, or has been rewritten by another value that is returned in some snapshot. To illustrate, in the above example, $\{5, 0, 2\} \subseteq \mathsf{scanned}\,\Omega$. Intuitively, we want to model that after a write has been observed, the ordering of the events logically preceding the write must be stabilized, and moreover, must be a sequence. Thus, $\mathsf{scanned}\,\Omega$

$$\texttt{write}\ (p, v) :\ \{\chi_{\mathsf{s}} = \varnothing\}\ \{\exists t.\, \chi'_{\mathsf{s}} = t \mapsto (p, v) \wedge \mathsf{dom}(\chi_{\mathsf{o}}) \cup \mathsf{scanned}\ \Omega \subseteq \Omega' \downdasharrow t\}@C$$

$$\texttt{scan} : \{\chi_{\mathsf{s}} = \varnothing\}\ \{r.\, \exists t.\, \chi'_{\mathsf{s}} = \varnothing \wedge r = \mathsf{eval}\ t\, \Omega'\, \chi' \wedge \mathsf{dom}(\chi) \subseteq \Omega' \downarrow t \wedge t \in \mathsf{scanned}\, \Omega'\}@C$$

Figure 4 Snapshot method specification.

is a *linearly ordered subset* of Ω.[2] The set $\mathsf{scanned}\,\Omega$ can also be seen as *representing all the scans that have already been executed.* Such representation of scans allows us to avoid tracking scan events directly in the history.

In the sequel, we concretize Ω and $\mathsf{scanned}\,\Omega$ in terms of σ and other auxiliary state. However, we keep the notions abstract in the method specs and in client reasoning. This enables the use of different snapshot algorithms, with the same specs, without invalidating the client proofs. We also mention that σ, Ω and $\mathsf{scanned}\,\Omega$ can be encoded as user-level concepts in FCSL, and require no new logic to be developed.

Snapshot specification. Figure 4 presents our specs for `scan` and `write`. These are partial correctness specs that describe how the methods change the state from the precondition (first braces) to the postcondition (second braces), possibly influencing the value r that the procedure returns. We use VDM-style notation with unprimed variables for the state before, and primed variables for the state after the method executes. We use Greek letters for state-dependent values that can be mutated by the method, and Latin letters for immutable variables. The component C is a state transition system (STS) that describes the state space of the algorithm, i.e, the invariants on the auxiliary and real state, and the transitions, i.e., the allowed atomic mutations of the state. For now, we keep C abstract, but will define it in Sections 5 and 6. We denote by $\Omega \downarrow t$ the downward-closed set of timestamps $\Omega \downarrow t = \{s \mid s\ \Omega\ t\}$. Let $\Omega \downdasharrow t = (\Omega \downarrow t) \backslash \{t\}$.

The spec for `write` says the following. The precondition starts with the empty self history χ_{s}, indicating that the procedure has not made any writes. In the postcondition, a new write event $t \mapsto (p, v)$ has been placed into χ'_{s}. Thus, a call to `write` wrote v into pointer p. The timestamp t is fresh, because χ' does not contain duplicate timestamps. Moreover, the write appears as if it occurred atomically at time t, thus capturing the logical atomicity of `write`.

The next conjunct, $\mathsf{dom}(\chi_{\mathsf{o}}) \cup \mathsf{scanned}\,\Omega \subseteq \Omega' \downdasharrow t$, positions the write t into the context of other events. In particular, if $s \in \mathsf{dom}(\chi_{\mathsf{o}})$, *i.e.*, if s finished prior to invoking `write`, then s is logically ordered strictly before t. In other words, `write` cannot reorder prior events that did not overlap with it. The definition of linearizability contains a similar prohibition on reordering non-overlapping events, but here, we capture it using a Hoare-style spec. For similar reasons, we require that $\mathsf{scanned}\,\Omega \subseteq \Omega' \downdasharrow t$. As mentioned before, $\mathsf{scanned}\,\Omega$ represents all the scans that finished prior to the call to `write`. Consequently, they do not overlap with `write` in real time, and have to be logically ordered before t.

Notice what the spec of `write` *does not prevent.* It is possible that some event, say with a timestamp s, finishes in real time before the call of `write` at time t. Events s and t do not overlap, and hence cannot be reordered; thus $s\ \Omega\ t$ always. However, the relationship of s with other events that ran concurrently with s, may be fixed only later, thus supporting implementation of "future-dependent" nature, such as Jayanti's.

[2] In terminology of linearizability, one may say that $\mathsf{scanned}\,\Omega$ is the set of "linearized" writes.

In the case of `scan`, we start and terminate with an empty χ_{s}, because `scan` does not create any write events, and we do not track scan events. However, when `scan` returns the pair $r = (r_x, r_y)$, we know that there exists a timestamp t that describes when the scan took place. This t is the timestamp of the last write preceding the call to `scan`.

The postcondition says that t is the moment in which the snapshot was logically taken, by the conjunct $r = \mathsf{eval}\ t\ \Omega'\ \chi'$. Here, eval is a pure, specification-level function that works as follows. First, it reorders the entire real-time post-history χ' according to logical post-ordering Ω'. Then, it computes and returns the values of x and y that would result from executing the write events of such reordered history up to the timestamp t. For example, if t is the timestamp of event 1 in Figure 3b, then $\mathsf{eval}\ t\ \Omega'\ \chi'$ would return $(2, 1)$. Hence, the conjunct says that `scan` performed a scan of x and y, consistent with the ordering Ω', and returned the read values into r. The scan appears as if it occurred atomically, immediately after time t, thus capturing the atomicity of `scan`.

The next conjunct, $\mathsf{dom}(\chi) \subseteq \Omega' \downarrow t$, says that the scanner returned a snapshot that is current, rather than corresponding to an outdated scan. For example, referring to Figure 3, if `scan` is invoked after the events 2 and 1 have already executed, then `scan` should not return the pair $(5, 0)$ and have t be the timestamp of the event 0, because that snapshot is outdated. Specifically, the conjunct says that the write events from χ are ordered no later than t, similar to the postcondition of `write`. However, while in `write` we constrained the events from $\mathsf{dom}(\chi_{\mathsf{o}}) \cup \mathsf{scanned}\,\Omega$, here we constrain the full global history $\chi = \chi_{\mathsf{o}} \cup \chi_{\mathsf{j}}$. The addition of χ_{j} shows that the scanner will observe and order all of the write events that have been timestamped and recorded in χ_{j} (and thus, that have written their value to memory), prior to the invocation of `scan`.

Lastly, the conjunct $t \in \mathsf{scanned}\,\Omega'$ explicitly says that t has been observed by the just finished call to scan.

Again, it is important what the spec does not prevent. It is possible that the timestamp t identified as the moment of the scan, corresponds to a write that has been initiated, but has not yet terminated. Despite being ongoing, t is placed into $\mathsf{scanned}\,\Omega'$ (*i.e.*, t is "linearized"). Also, notice that the postcondition of `scan` actually specifies the "linearization" order of events that are initiated by another method, namely `write`, thus supporting implementations of "non-local" nature, such as Jayanti's.

We close the section with a brief discussion of how the specs are used. Because C, Ω and $\mathsf{scanned}$ are abstracted from the clients, we need to provide an interface to work with them. The interface consists of a number of properties showing how various assertions interact, summarized in the statements below.

The first statement presents the invariants on the transitions of STS C, often referred to as 2-state invariants. Another way of working with such invariants is to include them in the postcondition of every method.[3] For simplicity, here we agglomerate the properties, and use them implicitly in proofs as needed.

▸ **Invariant 1** (Transition invariants)**.** In any program respecting the transitions of C:

1. $\chi \subseteq \chi'$, $\chi_{\mathsf{s}} \subseteq \chi'_{\mathsf{s}}$, and $\chi_{\mathsf{o}} \subseteq \chi'_{\mathsf{o}}$.
2. $\Omega \subseteq \Omega'$ and $\mathsf{scanned}\,\Omega \subseteq \mathsf{scanned}\,\Omega'$.
3. For every $s \in \mathsf{scanned}\,\Omega$, $\Omega \downarrow s = \Omega' \downarrow s$.

Invariant 1.1 says that histories only grow, but does not insist that $\chi_{\mathsf{j}} \subseteq \chi'_{\mathsf{j}}$, as timestamps can be removed from χ_{j} and transferred to χ_{s}. Invariant 1.2 states that Ω is monotonic, and

[3] In fact, this is what we currently do in our Coq files.

the same applies for $\mathsf{scanned}\,\Omega$. This is a fundamental stability requirement for our system: no transition in the STS C can change the relations between write events in Ω and, moreover, write events which have been observed by the scanner— and thus are in $\mathsf{scanned}\,\Omega$— cannot be unobserved. Invariant 1.3 says that if a new event is added to increase Ω to Ω', that event appears logically later than any $s \in \mathsf{scanned}\,\Omega$. In other words, once events are observed by a scanner, and placed into $\mathsf{scanned}\,\Omega$ in a certain order, we cannot insert new events among them to modify the past observation.

The second statement exposes the properties of Ω, $\mathsf{scanned}$, and eval that are used for client reasoning:

▸ **Invariant 2** (Relating $\mathsf{scanned}$ and snapshots)**.** The set $\mathsf{scanned}\ \Omega$ satisfies the following properties:

1. if $t_1 \in \mathsf{scanned}\,\Omega$ and $t_2 \in \mathsf{scanned}\,\Omega$, then $t_1\ \Omega\ t_2 \vee t_2\ \Omega\ t_1$ (linearity).
2. if $t_2 \in \mathsf{scanned}\,\Omega$ and $t_1\ \Omega\ t_2$, then $t_1 \in \mathsf{scanned}\,\Omega$ (downward closure).
3. if $t \in \mathsf{scanned}\,\Omega$, $\chi \subseteq \chi'$, $\Omega \subseteq \Omega'$, $\mathsf{scanned}\,\Omega \subseteq \mathsf{scanned}\,\Omega'$, and $\Omega \downarrow t = \Omega' \downarrow t$ then $\mathsf{eval}\,t\,\Omega\,\chi = \mathsf{eval}\,t\,\Omega'\,\chi'$. (snapshot preservation).

The first two properties merely state that the subset $\mathsf{scanned}\,\Omega$ is totally-ordered (2.1) and also downward closed (2.1). The last property is the most interesting: it entails that once a snapshot is observed by `scan`, its validity will not be compromised by future or ongoing calls to `write`. Thus, snapshots returned from previous calls to `scan` are still valid and observable in the future.

4 Client reasoning

Comparison with linearizability specifications. In linearizability one would specify `write` and `scan` by relating them, via a simulation argument, to sequential programs for writing and scanning, respectively. On the face of it, such specs are indeed simpler than ours above, as they merely state that `write` writes and `scan` scans. Our specs capture this property with one conjunct in each postcondition. The remainders of the postconditions describe the relative order of the atomic events, *observed* by threads, including explicit prohibition on reordering non-overlapping events, which is itself inherent in the definition of linearizability.

However, the additional specifications are not pointless, and they become useful when it comes to reasoning about clients. Linearizability tells us that we can simplify a fine-grained client program by replacing the occurrences of `write` and `scan` with the atomic and sequential equivalents, thus turning the client into an equivalent coarse-grained concurrent program. However, linearizability is not directly concerned with verifying that coarse-grained equivalent itself. Then, if one is interested in proving client properties which involve timing and/or ordering properties of such events, it is likely that the simple sequential spec described above do not suffice, and extra auxiliary state is still required.

On the other hand, if one wants to reason about such clients using a Hoare logic, then our specs are immediately useful. Moreover, in our setting, client reasoning depends solely on the API for scan and write, regardless of the different linearizations of a program. In the sequel, we illustrate this claim by deriving interesting client timing properties out of the specs of `write` and `scan`.

Moreover, because we use separation logic, our approach easily supports reasoning about programs with a dynamic number of threads, and about programs that transfer state ownership. In fact, as we already commented in Section 3, our proofs rely on transferring write events from χ_{j} (joint ownership) to χ_{s} (private ownership), upon `write`'s termination.

This is immediate in FCSL, as reasoning about histories inherits the infrastructure of the ordinary heap-based separation logic, such as framing and, in this case, ownership transfer. In contrast, Linearizability is usually considered for a fixed number of threads, and its relationship with ownership transfer is more subtle [4, 14].

An additional benefit of specifying the event orders by Hoare triples at the user level, is that one can freely combine methods with different event-ordering properties, that need not respect the constraints of linearizability [38].

Example clients. We first consider the client e, defined as follows:

$$\begin{array}{c} \texttt{write}\ (x, 2); \\ \texttt{write}\ (y, 1) \end{array} \;\Big\|\; \texttt{scan}\ () \;\Big\|\; \texttt{write}\ (x, 3)$$

It is our running example from Figure 2a. We will show that it satisfies the spec below. In the sequel we omit the STS C, as it never changes.

$$\begin{aligned} e : \{\chi_{\mathsf{s}} = \varnothing\}\, \{r.\, \exists t_1\, t_2\, t_3\, t_s.\ \chi'_{\mathsf{s}} = t_1 \mapsto (y, 1) \mathbin{\dot\cup} t_2 \mapsto (x, 2) \mathbin{\dot\cup} t_3 \mapsto (x, 3) \wedge \mathsf{dom}(\chi) \subseteq \Omega' \downarrow t_s \wedge \\ \mathsf{dom}(\chi_{\mathsf{o}}) \subseteq \Omega' \dashdownarrow t_2, \Omega' \dashdownarrow t_3 \wedge t_2\ \Omega'\ t_1 \wedge r = \mathsf{eval}\ t_s\ \Omega'\ \chi'\} \end{aligned}$$

The spec of e states that (1) `write` $(x, 2)$, timestamped t_2, occurs sequentially before `write` $(y, 1)$ which is timestamped t_1, (2) the remaining write, timestamped t_3, and the scan, timestamped t_s, are not temporally constrained, and (3) the writes that terminated before the client started are ordered before t_2 (and thus before t_1), t_3 and t_s. The example illustrates how to track timestamps and their order, but does not utilize the properties of $\mathsf{scanned}\,\Omega$. We illustrate the latter in another example at the end of this section.

We first verify the subprograms `scan` $()$ $\ \|$ `write` $(x, 3)$ and `write` $(x, 2)$; `write` $(y, 1)$ separately, and then combine them into the full proof. As proof outlines show intermediate, in addition to pre- and post-state, we cannot quite utilize VDM notation in them. As a workaround, we explicitly introduce logical variables h and h_o to name (subsets of) the initial global and other history.

```
1                          {$\chi_{\mathsf{s}} = \varnothing \wedge h \subseteq \chi \wedge h_o \subseteq \chi_{\mathsf{o}}$}
   2a  {$\chi_{\mathsf{s}} = \varnothing \wedge h \subseteq \chi \wedge h_o \subseteq \chi_{\mathsf{o}}$}      ||   2b  {$\chi_{\mathsf{s}} = \varnothing \wedge h \subseteq \chi \wedge h_o \subseteq \chi_{\mathsf{o}}$}
   3a              scan ()                    ||   3b           write $(x, 3)$
   4a  {$r.\ \exists t_s. \chi_{\mathsf{s}} = \varnothing \wedge \mathsf{dom}(h) \subseteq \Omega \downarrow t_s \wedge$     ||   4b  {$\exists t_3. \chi_{\mathsf{s}} = t_3 \mapsto (x, 3) \wedge$
              $r = \mathsf{eval}\, t_s\, \Omega\, \chi$}            ||            $\mathsf{dom}(h_o) \subseteq \Omega \dashdownarrow t_3$}
5  {$r.\ \exists t_3\, t_s.\ \chi_{\mathsf{s}} = t_3 \mapsto (x, 3) \wedge \mathsf{dom}(h_o) \subseteq \Omega \dashdownarrow t_3 \wedge \mathsf{dom}(h) \subseteq \Omega \downarrow t_s \wedge r = \mathsf{eval}\ t_s\ \Omega\ \chi$}
```

The proof applies the rule for parallel composition of FCSL. This rule is described in Appendix A. Here, we just mention that, upon forking, the rule distributes the value of χ_{s} of the parent thread, to the χ_{s} values of its children; in this case, all these are $\varnothing$. Dually, upon joining, the χ_{s} values of the children in lines 4a and 4b, are collected, in line 5, into that of the parent. The other assertions in 4a and 4b directly follow from the specs of `scan` and `write` and the Invariants 1.1 and 1.2, and directly transfer to line 5. While the proof outline does not establish how `scan` and `write` interleaved, it establishes that t_3 and t_s both appear after the writes that are prior to the client's call.

```
1  {$\chi_{\mathsf{s}} = \varnothing \wedge h \subseteq \chi \wedge h_o \subseteq \chi_{\mathsf{o}}$}
2  write $(x, 2)$;
3  {$\exists t_2.\ \chi_{\mathsf{s}} = t_2 \mapsto (x, 2) \wedge \mathsf{dom}(h_{\mathsf{o}}) \subseteq \Omega \dashdownarrow t_2$}
4  write $(y, 1)$
5  {$\exists t_1\, t_2.\ \chi_{\mathsf{s}} = t_1 \mapsto (y, 1) \mathbin{\dot\cup} t_2 \mapsto (x, 2) \wedge \mathsf{dom}(h_{\mathsf{o}}) \subseteq \Omega \dashdownarrow t_2 \wedge t_2\ \Omega\ t_1$}
```

The second proof outline starts with the same precondition. Then line 3 directly follows from the spec of `write`, using $h_o \subseteq \chi_{\mathsf{o}}$. To proceed, we need to apply FCSL *framing*: the precondition of `write` requires $\chi_{\mathsf{s}} = \varnothing$, but we have $\chi_{\mathsf{s}} = t_2 \mapsto (x, 2)$. The frame rule is explained in Appendix A. Here we just mention that framing modifies the spec of `write` by joining $t_2 \mapsto (x, 2)$ to χ_{s}, χ'_{s} *and* χ_{o} as follows.

$$\begin{aligned}\texttt{write}\ (p, v) :\ \ &\{\chi_{\mathsf{s}} = t_2 \mapsto (x, 2)\}\ \{\exists t.\, \chi'_{\mathsf{s}} = t \mapsto (p, v) \mathbin{\dot\cup} t_2 \mapsto (x, 2)\ \wedge \\ &\qquad\qquad \mathsf{dom}(\chi_{\mathsf{o}} \mathbin{\dot\cup} t_2 \mapsto (x, 2)) \cup \mathsf{scanned}\ \Omega \subseteq \Omega' \twoheaddownarrow t\}\end{aligned}$$

Such a framed spec for `write` gives us that after line 4: (1) $\chi_{\mathsf{s}} = t_1 \mapsto (y, 1) \mathbin{\dot\cup} t_2 \mapsto (x, 2)$, and (2) $\mathsf{dom}(h_{\mathsf{o}} \mathbin{\dot\cup} t_2 \mapsto (x, 2)) \subseteq \Omega \twoheaddownarrow t_1$. From Invariants 1, we also obtain that (3) $\mathsf{dom}(h_{\mathsf{o}}) \subseteq \Omega \twoheaddownarrow t_2$, which simply transfers from line 3. Now, in the presence of (2), we can simplify (3) into $t_2\ \Omega\ t_1$, thus obtaining the postcondition in line 5.

The final step applies the rule for parallel composition to the two derivations, splitting χ_{s} upon forking, and collecting it upon joining:

$$\begin{aligned}e :\ \ &\{\chi_{\mathsf{s}} = \varnothing \wedge h \subseteq \chi \wedge h_o \subseteq \chi_{\mathsf{o}}\} \\ &\{r.\, \exists\, t_1\, t_2\, t_3\, t_s.\, \chi_{\mathsf{s}} = t_1 \mapsto (y, 1) \mathbin{\dot\cup} t_2 \mapsto (x, 2) \mathbin{\dot\cup} t_3 \mapsto (x, 3) \wedge \mathsf{dom}(h) \subseteq \Omega \downarrow t_s\ \wedge \\ &\qquad\qquad \mathsf{dom}(h_{\mathsf{o}}) \subseteq \Omega \twoheaddownarrow t_2, \Omega \twoheaddownarrow t_3 \wedge t_2\ \Omega\ t_1 \wedge r = \mathsf{eval}\, t_s\, \Omega\, \chi\}\end{aligned}$$

From here, the VDM spec of e is derived by priming the Greek letters in the postcondition, and choosing $h = \chi$ and $h_o = \chi_{\mathsf{o}}$.

The spec of e can be further used in various contexts. For example, to recover the context from Section 2, where e is invoked with $x = 5$, $y = 0$, we can frame e wrt. $\chi_{\mathsf{s}} = t_5 \mapsto (x, 5) \mathbin{\dot\cup} t_0 \mapsto (y, 0)$ to make explicit the events that initialize x and y. Then, it is possible to derive in FCSL that if e executes without interference (*i.e.*, if $\chi = \chi_{\mathsf{o}} = \chi' = \chi'_{\mathsf{o}} = \varnothing$), then the result at the end must be $r \in \{(5, 0), (2, 0), (3, 0), (2, 1), (3, 1)\}$. As expected, $r \neq (5, 1)$, because the write of 2 sequentially precedes the write of 1.

We next illustrate the use of Invariants 2, which are required for clients that use `scan` in *sequential composition*. We consider the program

$$e' = r \leftarrow \texttt{scan};\ \texttt{write}\ (x, v);\ \textbf{return}\, r$$

and prove that e' can be ascribed the following spec:

$$e' : \{\chi_{\mathsf{s}} = \varnothing\}\quad \{\exists\, t_s\ t_x.\ \chi'_{\mathsf{s}} = t_x \mapsto (x, v) \wedge t_s \in \Omega' \twoheaddownarrow t_x \wedge r = \mathsf{eval}\ t_s\ \Omega'\ \chi'\}$$

The spec says that the write event (t_x) is subsequent to the scan (t_s), as one would expect. In particular, the snapshot r remains valid, *i.e.*, the write does not change the order Ω and history χ in a way that makes r cease to be a valid snapshot in Ω' and χ'. The proof outline follows, with the explanation of the critical steps.

```
1   $\{\chi_{\mathsf{s}} = \varnothing\}$
2   $r \leftarrow \texttt{scan};$
3   $\{\exists\, t_s, w'(= \Omega), h'(= \chi).\, \chi_{\mathsf{s}} = \varnothing \wedge t_s \in \mathsf{scanned}\, w' \wedge r = \mathsf{eval}\ t_s\ w'\ h'\}$
4   $\texttt{write}\ (x, v);$
5   $\{\exists\, t_s\, t_x.\ \chi_{\mathsf{s}} = t_x \mapsto (x, v) \wedge t_s \in \Omega \twoheaddownarrow t_x \wedge t_s \in \mathsf{scanned}\, \Omega \wedge r = \mathsf{eval}\ t_s\ \Omega\ \chi\}$
6   $\textbf{return}\ r$
```

Line 3 is a direct consequence of the spec of `scan`, where we omitted the conjunct $\mathsf{dom}(\chi) \subseteq \Omega' \downarrow t_s$, as we do not need it for the subsequent derivation. We also introduce explicit names w' and h' for the current values of Ω and χ. Now, to derive line 5, by the spec of `write`, we know there exists a timestamp t_x corresponding to the write, such that (1) $\chi_{\mathsf{s}} = t_x \mapsto (x, v)$, which is a conjunct in line 5, and also (2) $\mathsf{dom}(\chi_{\mathsf{o}}) \cup \mathsf{scanned}\, w' \subseteq \Omega ⤈ t_x$. Furthermore, (3) $t_s \in \mathsf{scanned}\, w'$, and (4) $r = \mathsf{eval}\ t_s\ w'\ h'$, simply transfer from line 3. From (2) and (3), we infer that $t_s \in \Omega ⤈ t_x$. To complete the derivation of line 5, it remains to show that $t_s \in \mathsf{scanned}\, \Omega$ and $r = \mathsf{eval}\ t_s\ \Omega\ \chi$. For this, we use (3), (4) and the Invariants 1 and 2, as follows. First, by Invariant 1.3, and because $t_s \in \mathsf{scanned}\, w'$, we get $w' \downarrow t_s = \Omega \downarrow t_s$. By Invariant 1.2, this gives us $t_s \in \mathsf{scanned}\, \Omega$ as well. By Invariant 1.1, $h' \subseteq \chi$, and then by Invariant 2.3, $r = \mathsf{eval}\ t_s\ w'\ h' = \mathsf{eval}\ t_s\ \Omega\ \chi$, completing the deduction of line 5.

Observe that the main role of $\mathsf{scanned}$ in proofs is to enable showing *stability* of values obtained by eval, using Invariant 2.3. The remaining Invariants 2.1 and 2.2 allow us to replace a number of conjuncts about $\mathsf{scanned}$ by a single one that expresses the membership of the largest timestamp in the current $\mathsf{scanned}$ set.

5 Internal auxiliary state

In order to verify the *implementations* of `write` and `scan`, we require further auxiliary state that does *not* feature in the specifications, and is thus hidden from the clients.

First, we track the point of execution in which `write` and `scan` are, but instead of line numbers, we use datatypes to encode extra information in the constructors. For example, the scanner's state is a triple (S_s, S_x, S_y). S_s is drawn from $\{\mathsf{S_{On}}, \mathsf{S_{Off}}\, t\}$. If $\mathsf{S_{On}}$, then the scanner is in lines 7–11 in Figure 1. If $\mathsf{S_{Off}}\ t$, the the scanner reached line 12 at "time" t, and is now in 13–17. S_x is a boolean bit, set when the scanner clears *fx* in line 8, and reset upon scanner's termination (dually for S_y and *fy*). Writers' state for x is tracked by the auxiliary W_x (dually, W_y). These are drawn from $\{\mathsf{W_{Off}}, \mathsf{New}\ t\, v, \mathsf{Fwd}\ t\, v, \mathsf{Done}\ t\, v\}$, where t marks the beginning of the write and v is the value written to pointer p. If $\mathsf{W_{Off}}$, then no write is in progress. If $\mathsf{New}\ t\, v$, then the writer is in line 2. If $\mathsf{Fwd}\ t\, v$, then b has been set in line 3, triggering forwarding. If $\mathsf{Done}\ t\, v$, the writer is free to exit.

Second, like in linearizability, we record the ending times of terminated events, using an auxiliary variable τ. τ is a function that takes a timestamp identifying the beginning of some event, and returns the ending time of that event, and is undefined if the event has not terminated. However, we do not *generate* fresh timestamps to mark event ending times. Instead, at the end of `write`, we simply read off the last used timestamp in χ, and use it as the ending time of `write`. This is a somewhat non-standard way of keeping time, but it suffices to prove that events t_1 and t_2 which are non-overlapping (*i.e.*, $\tau(t_1) < t_2$ or $\tau(t_2) < t_1$) are never reordered. The latter is required by the postconditions of `write` and `scan`, as we discussed in Section 3. Formally, the following is an *invariant* of the snapshot object; *i.e.*, a property of the state space of STS C from Figure 4, preserved by C's transition.

▸ **Invariant 3.** The logical order $<_\sigma$ preserves the real time order of non-overlapping events: $\forall t_1 \in \mathsf{dom}(\tau), t_2 \in \mathsf{dom}(\chi)$, if $\tau(t_1) < t_2$ then $t_1 <_\sigma t_2$.

Third, we track the rearrangement status of write events wrt. an ongoing *active* scan, by *colors*. A scan is *active* if it has cleared the forwarding pointers in lines 8 and 9, and is ready to read x and y. We keep the auxiliary variable κ, which is a function mapping each timestamp in χ to a color, as follows.

- Green timestamps identify write events whose position in the logical order is fixed in the following sense: if $\kappa(t_1) = \mathsf{green}$ and $t_1 <_\sigma t_2$, then $t_1 <_{\sigma'} t_2$ for every σ' to which σ may step by auxiliary code execution (Section 6). For example, since we only reorder overlapping events, and only the scanner reorders events, every event that finished before the active scan started will be green. Also, a green timestamp never changes its color.

- Red timestamps identify events whose order is not fixed, but which will *not* be manipulated by the active scan, and are left for the next scan.
- Yellow timestamps identify events whose order is not fixed yet, but which *may* be manipulated by the ongoing active scan, as follows. The scan can *push* a yellow timestamp in logical time, *past* another green or yellow timestamp, but not past a red one. *This is the only way the logical ordering can be modified.*

There are a number of invariants that relate colors and timestamps. We next list the ones that are most important for understanding our proof. We use χ_p to denote the sequence of writes into the pointer p that appear in the history χ, sorted by their order in σ[4].

▸ **Invariant 4** (Colors)**.** The colors of χ_p are described by the regular expression $\mathbf{g}^+\mathbf{y}^?\mathbf{r}^*$: there is a non-empty prefix of green timestamps, followed by *at most* one yellow, and arbitrary number of reds.

By the above invariant, the yellow color identifies the write event into the pointer p, that is the *unique* candidate for reordering by the ongoing active scan. Moreover, all the writes into p prior to the yellow write, will have already been colored green (and thus, fixed in time), whether they overlapped with the scanner or not.

▸ **Invariant 5** (Color of forwarded values)**.** Let $S_s = \mathsf{Off}\ t_{\mathsf{off}}$, and $p \in \{x, y\}$, and $S_p = \mathsf{True}$ (*i.e.*, scanner is in lines 13–16), and $v \neq \bot$ has been forwarded to p; *i.e.*, $\texttt{fwd}\ p \mapsto v$. Then the event of writing v into p is in the history, *i.e.*, there exists t such that $t \mapsto (p, v) \in \chi_p$. Moreover, t is the last green, or the yellow timestamp in χ_p.

The above invariant restricts the set of events that could have forwarded a value to the scanner, to only two: the event with the (unique) yellow timestamp, or the one corresponding to the last green timestamp. By Invariant 4, these two timestamps are consecutive in χ_p.

▸ **Invariant 6** (Red zone)**.** If $S_s = \mathsf{Off}\ t_{\mathsf{off}}, S_x = \mathsf{True}, S_y = \mathsf{True}$, then χ satisfies the $(\mathbf{g}|\mathbf{y})^+\mathbf{r}^*$ pattern. Moreover, for every $t \in \mathsf{dom}(\chi)$:

- $\kappa(t) = \mathsf{green} \implies t \leqslant t_{\mathsf{off}}$
- $\kappa(t) = \mathsf{yellow} \implies t \leqslant t_{\mathsf{off}} \leqslant \tau(t)$
- $\kappa(t) = \mathsf{red} \implies t_{\mathsf{off}} < t$

This invariant restricts the global history χ (not the pointer-wise projections χ_p). First, the red events in χ are consecutive, and cannot be interspersed among green and yellow events. Thus, when a scanner pushes a yellow event past a green event, or past another yellow event, it will not “jump over” any reds. Second, the invariant relates the colors to the time t_{off} at which the scanner was turned off (in line 12, Figure 1). This moment is important for the algorithm; *e.g.*, it is the linearization point for `scan` in Jayanti's proof [22]. We will use the above inequalities wrt. t_{off} in our proofs, to establish that the events reordered by the scanner *do* overlap, as per Invariant 3.

We can now define the stable logical order Ω, and the set $\mathsf{scanned}\,\Omega$, using the internal auxiliary state of colors and ending times.

[4] For reasoning purposes, it serves us better to think of χ_p as sub-histories, with an *e*xternal ordering given by σ. We do, however, implement χ_p as a list filter: $\chi_p = \mathsf{filter}\ (\lambda\, t.\, t \mapsto (p, \_) \in \chi)\ \sigma$.

```
1    write $(p,\ v)$ {
2      $\langle\, p := v;\ register(p, v)\rangle$;
3      $\langle\, b \leftarrow \mathsf{read}(S);\ \ check(p, b)\rangle$;
4      if $b$
5      then $\langle\,$fwd $p := v;\ \ forward(p)\rangle$;
5'     $\langle\, finalize(p)\rangle$}

6    scan() : $(A \times A)$ {
7      $\langle\, S :=$ true; $set($true$)\rangle$;
8      $\langle\, fx := \bot;\ \ clear(x)\rangle$;
9      $\langle\, fy := \bot;\ \ clear(y)\rangle$;
10     $vx \leftarrow \langle \mathsf{read}(x)\rangle$;
11     $vy \leftarrow \langle \mathsf{read}(y)\rangle$;
12     $\langle\, S :=$ false; $set($false$)\rangle$;
13     $ox \leftarrow \langle \mathsf{read}(fx)\rangle$;
14     $oy \leftarrow \langle \mathsf{read}(fy)\rangle$;
15     $rx \leftarrow$ if $(ox \neq \bot)$ then $ox$ else $vx$;
16     $ry \leftarrow$ if $(oy \neq \bot)$ then $oy$ else $vy$;
17     $\langle\, relink(rx, ry);$ return $(rx, ry)\,\rangle$}
```

Figure 5 Snapshot procedures annotated with auxiliary code.

▸ **Definition 7** (Logical order Ω and $\mathsf{scanned}\,\Omega$). **1.** $t_1 \,\Omega\, t_2 \;\hat{=}\; (t_1 = t_2) \vee (\tau(t_1) < t_2) \vee (t_1 <_\sigma t_2 \wedge \kappa(t_1) = \mathsf{green})$

2. $\mathsf{scanned}\,\Omega = \{t \mid \Omega \downarrow t = \leqslant_\sigma \downarrow t \wedge \forall s \in \Omega \downarrow t.\, \kappa(s) = \mathsf{green}\}$.

From the definition of Ω, notice that $t_1 \,\Omega\, t_2$ is stable (*i.e.*, invariant under interference), since threads do not change the ending times τ, the color of green events, or the order of green events in $<_\sigma$, as we already discussed. From the definition of $\mathsf{scanned}\,\Omega$, notice that for every $t \in \mathsf{scanned}\,\Omega$, it must be that $\Omega \downarrow t$ is a linearly-ordered set wrt. Ω, because it equals a prefix of the *sequence* σ.

We close this section with a few technical invariants that we use in the sequel.

▸ **Invariant 8** (Last write). Let pointer $p \in \{x,\ y\}$, and $\mathsf{last}_\sigma\ \chi_p$ be the timestamp in χ_p that is largest wrt. the logical order $\leqslant_\sigma$. Then the contents of p equals the value written by the event associated with $\mathsf{last}_\sigma\ \chi_p$. That is, $p \mapsto \chi_p(\mathsf{last}_\sigma \chi_p)$.

▸ **Invariant 9** (Joint history). Let pointer $p \in \{x,\ y\}$. If the writer for p is active *i.e.* $W_p \neq \mathsf{W_{Off}}$, then the write event that it is performing is timestamped and placed into joint history χ_{j}. Dually, if $t \in \mathsf{dom}(\chi_{\mathsf{j}})$, then the event t is performed by the active writer for p:

$$t \mapsto (p, v) \in \chi_{\mathsf{j}} \iff W_p = \mathsf{New}\ t\, v \vee W_p = \mathsf{Fwd}\ t\, v \vee W_p = \mathsf{Done}\ t\, v$$

▸ **Invariant 10** (Terminated events). Histories χ_{o} and χ_{s} store only terminated events, *i.e.*, events whose ending times are recorded in τ. Moreover, the codomain of τ is bounded by the maximal timestamp, in real time, in $\mathsf{dom}(\chi)$:

1. $\mathsf{dom}(\tau) = \mathsf{dom}(\chi_{\mathsf{s}}) \cup \mathsf{dom}(\chi_{\mathsf{o}})$.

2. $\forall a \in \mathsf{dom}(\tau).\ \ \tau(a) \leqslant \mathsf{max}\ (\mathsf{dom}(\chi))$.

▸ **Lemma 11** (Green/yellow read values). *Let $p \in \{x, y\}$. If the scanner state is $S_s = \mathsf{S_{On}}, S_p =$ True, i.e., the scanner is between lines 10–11 in Figure 1, and $p \mapsto v$ in the physical heap, then exists t such that $t \mapsto (p, v) \in \chi_p$. Moreover, t is the last green or the yellow timestamp in χ_p.*

▸ **Lemma 12** (Chain). *If $t \in \mathsf{dom}(\chi)$ and $\kappa(\leqslant_\sigma \downarrow t) = \mathsf{green}$, then $\Omega \downarrow t \;=\; \leqslant_\sigma \downarrow t$.*

$$
\begin{array}{lll}
\mathit{register}(p,v): & \multicolumn{2}{l}{\langle W_p = \mathsf{W_{Off}}\rangle} \\
 & \multicolumn{2}{l}{\langle \sigma' = \mathsf{snoc}\ \sigma\ t,\ \chi'_{\mathsf{j}} = \chi_{\mathsf{j}} \mathbin{\dot\cup} t \mapsto (p,v),\ W'_p = \mathsf{New}\ t\, v,} \\
 & \multicolumn{2}{l}{\ \kappa' = \text{if } (S_s = \mathsf{S_{On}})\&S_p \text{ then } \kappa[t \mapsto \mathsf{yellow}] \text{ else } \kappa[t \mapsto \mathsf{red}]\rangle} \\
 & \multicolumn{2}{l}{\ \text{where } t = \mathsf{fresh}\ \chi = \mathsf{max}\,(\mathsf{dom}(\chi)) + 1} \\
\mathit{check}\,(p,b) & :[t,v]. & \langle W_p = \mathsf{New}\ t\, v\rangle\, \langle W'_p = \text{if } b \text{ then } \mathsf{Fwd}\ t\ v \text{ else } \mathsf{Done}\ t\, v\rangle \\
\mathit{forward}\,(p) & :[t,v]. & \langle W_p = \mathsf{Fwd}\ t\, v\rangle \\
 & & \langle W'_p = \mathsf{Done}\ t\, v,\ \kappa' = \text{if } (S_s = \mathsf{S_{On}})\&S_p \text{ then } \kappa[t \mapsto \mathsf{green}] \text{ else } \kappa\rangle \\
\mathit{finalize}(p) & :[t,v]. & \langle W_p = \mathsf{Done}\ t\, v, t \mapsto (p,v) \in \chi_{\mathsf{j}}\rangle \\
 & & \langle W'_p = \mathsf{W_{Off}},\ \chi'_{\mathsf{s}} = \chi_{\mathsf{s}} \mathbin{\dot\cup} t \mapsto (p,v),\ \chi'_{\mathsf{j}} = \chi_{\mathsf{j}} \setminus \{t\},\ \tau' = \tau \mathbin{\dot\cup} t \mapsto \mathsf{max}\,(\mathsf{dom}(\chi))\rangle
\end{array}
$$

$$
\begin{array}{lll}
\mathit{set}(b) & : & \langle S_s = \text{if } b \text{ then } \mathsf{S_{Off}}\ (\_) \text{ else } \mathsf{S_{On}},\ S_x = \neg\, b, S_y = \neg\, b\rangle \\
 & & \langle S'_s = \text{if } b \text{ then } \mathsf{S_{On}} \text{ else } \mathsf{S_{Off}}\ (\mathsf{last}\ \chi),\ S'_x = \neg\, b, S'_y = \neg\, b\rangle \\
\mathit{clear}(p) & : & \langle S_s = \mathsf{S_{On}},\ S_p = \mathsf{False}\rangle \\
 & & \langle S'_s = \mathsf{S_{On}},\ S'_p = \mathsf{True},\ \kappa' = \kappa[\chi_p \mapsto \mathsf{green}]\rangle \\
\multicolumn{3}{l}{\mathit{relink}(r_x,r_y) : [t_x,t_y].\ \ \langle S_s = \mathsf{S_{Off}}(\_), t_x \mapsto (x,r_x), t_y \mapsto (y,r_y) \in \chi, S_x = S_y = \mathsf{True},} \\
\multicolumn{3}{l}{\qquad\qquad\qquad\qquad\quad \forall p \in \{x,y\}.\, \mathsf{lastGY}\ p\, t_p\rangle} \\
\multicolumn{3}{l}{\qquad\qquad\qquad\qquad\ \ \langle S'_s = S_s, S'_x = S'_y = \mathsf{False}, \kappa' = \kappa[t_x, t_y \mapsto \mathsf{green}],} \\
\multicolumn{3}{l}{\qquad\qquad\qquad\qquad\quad \sigma' = \text{if } (d = \mathsf{Yes}\ x\ s) \text{ then } \mathit{push}\ s\ t_y\ \sigma} \\
\multicolumn{3}{l}{\qquad\qquad\qquad\qquad\qquad\quad \text{else if } (d = \mathsf{Yes}\ y\ s) \text{ then } \mathit{push}\ s\ t_x\ \sigma \text{ else } \sigma\rangle} \\
\multicolumn{3}{l}{\qquad\qquad\qquad\qquad\quad\ \text{where } d = \mathit{inspect}\ t_x\, t_y\, \sigma\, \kappa}
\end{array}
$$

Figure 6 Auxiliary procedures for `write` and `scan`. Bracketed variables (*e.g.*, $[t, v]$) are logical variables that scope over precondition and postcondition.

6 Auxiliary code implementation

Figure 5 annotates Jayanti's procedures with auxiliary code (typed in *italic*), with ⟨angle braces⟩ denoting that the enclosed real and auxiliary code execute *simultaneously* (*i.e.*, atomically). The auxiliary code builds the histories, evolves the sequence σ, and updates the color of various write events, while respecting the invariants from Section 3. Thus, it is the *constructive* component of our proofs. Each atomic command in Figure 5 represents one *transition* of the STS C from Figure 4.

The auxiliary code is divided into several procedures, all of which are sequences of reads followed by updates to auxiliary variables. We present them as Hoare triples in Figure 6, with the unmentioned state considered unchanged. The bracketed variables preceding the triples (*e.g.*, $[t, v]$) are logical variables used to show how the pre-state value of some auxiliary changes in the post-state. To symbolize that these triples *define* an atomic command, rather than merely stating the command's properties, we enclose the pre- and postcondition in angle brackets $\langle - \rangle$.

Auxiliary code for `write`. In line 2, $\mathit{register}(p, v)$ creates the write event for the assignment of v to p. It allocates a *fresh* timestamp t, inserts the entry $t \mapsto (p, v)$ into χ_{j}, and adds t to the end of σ, thus registering t as the currently latest write event. The fresh timestamp t is computed out of the history χ; we take the largest natural number occurring as a timestamp in χ, and increment it by 1. The variable W_p updates the writer's state to indicate that the writer finished line 2 with the timestamp t allocated, and the value v written into p. The color of t is set to yellow (*i.e.*, the order of t is left undetermined), but only if $(S_s = \mathsf{S_{On}})\&S_p$ (*i.e.*, an active scanner is in line 10). Otherwise, t is colored red, indicating that the order of t will be determined by a future scan.

In line 3, *check*(p, b), depending on b, sets the writer state to Fwd, indicating that a scan is in progress, and the writer should forward, or to Done, indicating that the writer is ready to terminate.

In line 5, *forward* colors the allocated timestamp t green, if an active scanner has passed lines 8–9 and is yet to reach line 12, because such a scanner will definitely see the write, either by reading the original value in lines 10–11, or by reading the forwarded value in lines 13–14. Thus, the logical order of t becomes fixed. In fact, it is possible to derive from the invariants in Section 3, that this order is the same one t was assigned at registration, *i.e.*, the linearization point of this write is line 2.

In line 5', *finalize* moves the write event t from the joint history χ_{j} to the thread's self history χ_{s}, thus acknowledging that t has terminated. The currently largest timestamp of χ is recorded in τ as t's ending time. By definition of Ω, all the writes that terminated before t in real time, will be ordered before t in Ω.

Auxiliary code for `scan`. Method *set* toggles the scanner state S_s on and off. When executed in line 12, it returns the timestamp t_{off} that is currently maximal in real time, as the moment when the scanner is turned off.

The procedure *clear*(p) is executed in lines 8–9 simultaneously with clearing the forwarding pointer for p. In addition to recording that the scanner passed lines 8 or respectively 9, by setting the S_p bit, it colors the subhistory χ_p green. Thus, by definition of $\mathsf{scanned}\,\Omega$, the ongoing one and all previous writes to p are recorded as scanned, and thus linearized.

Finally, the key auxiliary procedure of our approach is *relink*. It is executed at line 17 just before the scanner returns the pair (r_x, r_y). Its task is to modify the logical order of the writes, to make (r_x, r_y) *appear* as a valid snapshot. This will always be possible under the precondition of *relink* that the timestamps t_x, t_y of the events that wrote r_x, r_y respectively, are either the last green or the yellow ones in the respective histories χ_x and χ_y, and *relink* will consider all four cases. This precondition holds after line 16 in Figure 5, as one can prove from Invariants 4 and 5. In the precondition we introduce the following abbreviation:

$$\mathsf{lastGY}\ p\,t \ \ \widehat{=}\ \ t = \mathsf{last\_green}_\sigma\,\chi_p \vee \kappa(t) = \mathsf{yellow} \tag{2}$$

Relink uses two helper procedures *inspect* and *push*, to change the logical order. *Inspect* decides if the selected t_x and t_y determine a valid snapshot, and *push* performs the actual reordering. The snapshot determined by t_x and t_y is valid if there is no event s such that $t_x <_\sigma s <_\sigma t_y$ and s is a write to x (or, symmetrically $t_y <_\sigma s <_\sigma t_x$, and s is a write to y). If such s exists, *inspect* returns Yes x s (or Yes y s in the symmetric case). The reordering is completed by *push*, which moves s right after t_y (after t_x in the symmetric case) in $\leqslant_\sigma$. Finally, *relink* colors t_x and t_y green, to fix them in Ω. We can then prove that (r_x, r_y) is a valid snapshot wrt. Ω, and remains so under interference. Notice that the timestamp s returned by *inspect* is always uniquely determined, and yellow. Indeed, since t_x and t_y are not red, no timestamp between them can be red either (Invariant 6). If $t_x <_\sigma s <_\sigma t_y$ and s is a write to x (and the other case is symmetric), then t_x must be the last green in χ_x, forcing s to be the unique yellow timestamp in χ_x, by Invariant 4.

To illustrate, in Figure 3a we have $r_x = 2$, $r_y = 1$, t_x and t_y are both the last green timestamp of χ_x and χ_y, respectively, and $t_x <_\sigma t_y$. However, there is a yellow timestamp s in χ_x coming after t_x, encoding a write of 3. Because $t_x <_\sigma s <_\sigma t_y$, the pair (r_x, r_y) is not a valid snapshot, thus *inspect* returns Yes x s, after which *push* moves 3 after 1.

We have omitted the definitions of *inspect* and *push* for the sake of brevity. These are presented in Appendix B. We conclude this section with the main property of *relink*, whose proof can be found in our Coq files [1].

```
1   $\{\chi_{\mathsf{s}} = \emptyset \wedge w \subseteq \Omega \wedge h \subseteq \chi \wedge h_o \subseteq \chi_{\mathsf{o}}\}$
2   $\{\chi_{\mathsf{s}} = \emptyset \wedge W_p = \mathsf{W}_{\mathsf{Off}} \wedge w \subseteq \Omega \wedge h \subseteq \chi \wedge h_o \subseteq \chi_{\mathsf{o}}\}$
3   $\langle\, p := v;\, \mathit{register}(v) \rangle;$
4   $\{\exists t\,.\; \chi_{\mathsf{s}} = \emptyset \wedge W_p = \mathsf{New}\ t\ v \wedge t \mapsto (p, v) \in \chi_{\mathsf{j}} \wedge \mathsf{dom}(h_{\mathsf{o}}) \cup \mathsf{scanned}\, w \subseteq \Omega \downarrow t\}$
5   $\langle\, b \leftarrow \mathsf{read}(S);\;\; \mathit{check}(p, b) \rangle;$
6   $\{\exists t.\, \chi_{\mathsf{s}} = \emptyset \wedge W_p = \text{if } b \text{ then } \mathsf{Fwd}\ t\ v \text{ else } \mathsf{Done}\ t\ v \wedge\;\; t \mapsto (p, v) \in \chi_{\mathsf{j}} \wedge$
        $\mathsf{dom}(h_{\mathsf{o}}) \cup \mathsf{scanned}\, w \subseteq \Omega \downarrow t\}$
7   $\textbf{if } b \textbf{ then } \langle\, \texttt{fwd}\ p := v;\;\; \mathit{forward}(p, v) \rangle;$
8   $\{\exists t.\, \chi_{\mathsf{s}} = \emptyset \wedge W_p = \mathsf{Done}\ t\ v \wedge t \mapsto (p, v) \in \chi_{\mathsf{j}} \wedge \mathsf{dom}(h_{\mathsf{o}}) \cup \mathsf{scanned}\, w \subseteq \Omega \downarrow t\}$
9   $\langle\, \mathit{finalize}(i, v) \rangle$
10  $\{\exists t.\, \chi_{\mathsf{s}} = t \mapsto (p, v) \wedge \mathsf{dom}(h_{\mathsf{o}}) \cup \mathsf{scanned}\, w \subseteq \Omega \downarrow t\}$
```

Figure 7 Proof outline for `write`.

▸ **Lemma 13** (Main property of *relink*). *Let the precondition of relink hold,* i.e., $S_s = \mathsf{S}_{\mathsf{Off}}(\_)$, $t_x \mapsto (x, r_x), t_y \mapsto (y, r_y) \in \chi$, $S_x = S_y = \mathsf{True}$, *and* $\forall p \in \{x, y\}.\, \mathsf{lastGY}\ p\ t_p$. *Then the ending state of relink satisfies the following:*

1. *For all* $p \in \{x, y\}$, $t_p = \mathsf{last\_green}_{\sigma'}\, \chi'_p$.
2. *Let* $t = \mathsf{max}_{\sigma'}(t_x, t_y)$. *Then for every* $s \leqslant_{\sigma'} t$, $\kappa'(s) = \mathsf{green}$.

7 Correctness

We can now show that `write` and `scan` satisfy the specifications from Figure 4. As before, we avoid VDM notation in proof outlines by using logical variables.

Proof outline for `write` The proof outline for `write` is presented in Figure 7. Line 1 introduces logical variables w, h and h_o, which name the initial values of Ω, χ, and χ_{o}. Line 2 adds the knowledge that the writer for the pointer p is turned off ($W_p = \mathsf{W}_{\mathsf{Off}}$). This follows from our implicit assumption that there is only one writer in the system, which, in the Coq code, we enforce by locks.

Line 3 is the first command of the program, and the most important step of the proof. Here *register* allocates a fresh timestamp t for the write event, puts t into χ_{j}, coloring it yellow or red, and changes W_p to $\mathsf{New}\ t\ v$, simultaneously with the physical update of p with v (see Figure 6). The importance of the step shows in line 4, where we need to establish that t is placed into the logical order after all the other finished or scanned events (*i.e.*, $\mathsf{dom}(h_{\mathsf{o}}) \cup \mathsf{scanned}\ \Omega \subseteq \Omega \downarrow t$). This information is the most difficult part of the proof, but once established, it merely propagates through the proof outline.

Why does this inclusion hold? From the definition, we know that *register* appends t to the end of the list σ (the clause $\sigma' = \mathsf{snoc}\ \sigma\ t$ in the definition of *register* in Figure 6). Thus, after the execution of line 3, we know that for every other timestamp s, $s <_\sigma t$. In particular, $s \neq t$, so it suffices to prove $s\ \Omega\ t$. We consider two cases: $s \in \mathsf{dom}(h_o)$ and $s \in \mathsf{scanned}\,\Omega$. In the first case, by Invariant 10, $s \in \mathsf{dom}(\tau)$. By freshness of t wrt. global history h (which includes h_o), we get $\tau(s) < t$, and then the desired $s\ \Omega\ t$ follows from the definition of Ω. In the second case, by definition of $\mathsf{scanned}$, $\kappa(s) = \mathsf{green}$. Since $s <_\sigma t$, the result again follows by definition of Ω.

Still regarding line 4, we note that $t \in \mathsf{dom}(\chi_{\mathsf{j}})$ holds despite the interference of other threads. This is ensured by the Invariant 9, because no other thread but the writer for p, can modify W_p. Thus, this property will continue to hold in lines 6 and 8.

In line 6, the writer state W_p is updated following the definition of the auxiliary procedure *check*. The conjunct on $\mathsf{dom}(h_o) \cup \mathsf{scanned}\, w \subseteq \Omega \downarrow t$ propagates from line 4, by monotonicity of Ω (Invariant 1). Similarly, in line 8, W_p is changed following the definition of *forward*, and the the other conjunct propagates. *Forward* further colors a number of timestamps green, but this is done in order to satisfy the state space invariants from Section 3, and is not exposed in the proof of `write`. Finally, in line 10, *finalize* moves $t \mapsto (p, v)$ from χ_{j} to χ_{s}, thus completing the proof.

```
1   $\{\chi_{\mathsf{s}} = \varnothing \wedge h \subseteq \chi\}$
2   $\{\chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{Off}}\ \_ \wedge S_x = S_y = \mathsf{False} \wedge h \subseteq \chi\}$
3   $\langle\, S := \texttt{true};\ \mathit{set}(\texttt{true}) \rangle;$
4   $\{\,\chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{On}} \wedge S_x = S_y = \mathsf{False} \wedge h \subseteq \chi\}$
5   $\langle\, \mathit{fx} := \bot;\ \mathit{clear}(x) \rangle;$
6   $\{\chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{On}} \wedge S_x = \mathsf{True} \wedge S_y = \mathsf{False} \wedge h \subseteq \chi \wedge \kappa(\mathsf{dom}(h_x)) = \mathsf{green}\}$
7   $\langle\, \mathit{fy} := \bot;\ \mathit{clear}(y) \rangle;$
8   $\{\chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{On}} \wedge S_x = S_y = \mathsf{True} \wedge h \subseteq \chi \wedge \kappa(\mathsf{dom}(h)) = \mathsf{green}\}$
9   $\mathit{vx} \leftarrow \langle \mathsf{read}(x) \rangle;$
10  $\{\exists\, t_x.\ \chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{On}} \wedge S_x = S_y = \mathsf{True} \wedge$
        $h \subseteq \chi \wedge \kappa(\mathsf{dom}(h)) = \mathsf{green} \wedge \mathsf{fwdLastGY}\ x\, t_x\, \mathit{vx}\}$
11  $\mathit{vy} \leftarrow \langle \mathsf{read}(y) \rangle;$
12  $\{\exists\, t_x\, t_y.\ \chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{On}} \wedge S_x = S_y = \mathsf{True} \wedge h \subseteq \chi \wedge$
          $\kappa(\mathsf{dom}(h)) = \mathsf{green} \wedge \mathsf{fwdLastGY}\ x\, t_x\, \mathit{vx} \wedge \mathsf{fwdLastGY}\ x\, t_x\, \mathit{vy}\}$
13  $\langle\, S := \texttt{false};\ \mathit{set}(\texttt{false}) \rangle;$
14  $\{\exists\, t_x\, t_y\, t_{\mathsf{off}}.\ \chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}} \wedge S_x = S_y = \mathsf{True} \wedge h \subseteq \chi \wedge$
              $\kappa(\mathsf{dom}(h)) = \mathsf{green} \wedge \mathsf{fwdLastGY}\ x\, t_x\, \mathit{vx} \wedge \mathsf{fwdLastGY}\ y\, t_y\, \mathit{vy}\}$
15  $\mathit{ox} \leftarrow \langle \mathsf{read}(\mathit{fx}) \rangle;$
16  $\{\,\exists\, t_y\, t'_x\, t_{\mathsf{off}}.\ \chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}} \wedge S_x = S_y = \mathsf{True} \wedge h \subseteq \chi \wedge$
              $\kappa(\mathsf{dom}(h)) = \mathsf{green} \wedge \mathsf{fwdLastGY}\ y\, t_y\, \mathit{vy} \wedge$
              $\mathsf{lastGYHist}\ x\ t'_x\ (\text{if } r = \bot \text{ then } \mathit{vx} \text{ else } r)\}$
17  $\mathit{oy} \leftarrow \langle \mathsf{read}(\mathit{fy}) \rangle;$
18  $\{\,\exists\, t'_x\, t'_y\, t_{\mathsf{off}}.\ \chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}} \wedge S_x = S_y = \mathsf{True} \wedge h \subseteq \chi \wedge$
              $\kappa(\mathsf{dom}(h)) = \mathsf{green} \wedge \mathsf{lastGYHist}\ x\ t'_x\ (\text{if } \mathit{ox} = \bot \text{ then } \mathit{vx} \text{ else } \mathit{ox}) \wedge$
              $\mathsf{lastGYHist}\ y\ t'_y\ (\text{if } \mathit{oy} = \bot \text{ then } \mathit{vy} \text{ else } \mathit{oy})\}$
19  $\mathit{rx} \leftarrow \textbf{if}\ (\mathit{ox} \neq \bot)\ \textbf{then}\ \mathit{ox}\ \textbf{else}\ \mathit{vx};$
20  $\mathit{ry} \leftarrow \textbf{if}\ (\mathit{oy} \neq \bot)\ \textbf{then}\ \mathit{oy}\ \textbf{else}\ \mathit{vy};$
21  $\{\,\exists\, t'_x\, t'_y\, t_{\mathsf{off}}.\ \chi_{\mathsf{s}} = \varnothing \wedge S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}} \wedge S_x = S_y = \mathsf{True} \wedge h \subseteq \chi \wedge$
              $\kappa(\mathsf{dom}(h)) = \mathsf{green} \wedge \mathsf{lastGYHist}\ x\ t'_x\ \mathit{rx} \wedge \mathsf{lastGYHist}\ y\ t'_y\ \mathit{ry}\}$
22  $\langle\, \mathit{relink}(\mathit{rx}, \mathit{ry});\ \textbf{return}\ (\mathit{rx}, \mathit{ry}) \,\rangle$
23  $\{\, r.\ \exists t.\ \chi_{\mathsf{s}} = \varnothing \wedge r = \mathsf{eval}\ t\ \Omega\ \chi \wedge \mathsf{dom}(h) \subseteq \Omega \downarrow t \wedge t \in \mathsf{scanned}\ \Omega\}$
```

Figure 8 Proof outline for `scan`.

Proof outline for `scan`. Finally, the proof outline for is given in Figure 8. Line 1 introduces the logical variable h to name the initial χ. Line 2 adds the knowledge that $S_s = \mathsf{S_{Off}}\ \_$ and

$S_x = S_y = \mathsf{False}$, *i.e.*, that there are no other scanners around, which is enforced by locking in our Coq files.

Line 3 is the first line of the code; it simply sets the scanner bit S, and the auxiliaries S_x and S_y, following the definition of *set*. The conjunct $h \subseteq \chi$ follows from monotonicity by Invariant 1. The first important property comes from the lines 5 and 7. In these lines, *clear* sets the values of S_x and S_y, but, importantly, also colors the events from h green, first coloring x-events, and then y-events. This will be important at the end of the proof, where the fact that h is all green will enable inferring the postcondition. Moreover, because green events are never re-colored, we propagate this property to subsequent lines without commentary.

The read from x in line 9, and from y in line 11, must return the last green, or the yellow event of their pointer, if no values are forwarded in *fx* and *fy*, respectively. This holds by Lemma 11, and is reflected by the conjuncts $\mathsf{fwdLastGY}\ x\ t_x\ vx$ and $\mathsf{fwdLastGY}\ x\ t_x\ vy$ in line 12, where:

$$\mathsf{fwdLastGY}\ p\ t\ v\ \mathrel{\widehat{=}}\ \mathtt{fwd}\ p \mapsto \bot\ \implies\ \mathsf{lastGY}\ p\ t \wedge t \mapsto (p, v) \in \chi$$

The implication guard $\mathtt{fwd}\ p \mapsto \bot$ will be stripped away in the future, if and when the reads of forwarding pointers in lines 15 and 17 observe that no forwarding values exist.

In line 13, the scanner unsets the bit S and records the ending time of the scanner into the variable t_{off} in line 14. The conjuncts $\mathsf{fwdLastGY}\ x\ t_x\ vx$ and $\mathsf{fwdLastGY}\ y\ t_y\ vy$ from line 12 transfer to line 14 directly. This is so because *set* does not change any colors. Moreover, any writes that may run concurrently with this scan cannot invalidate the conjuncts. To see this, assume that we had a concurrent `write` to x (reasoning is symmetric for y). Such a `write` may add a new yellow timestamp s, but only if t_x itself is the last green, in accord with Invariant 4. In that case, t_x remains the last green timestamp, and $\mathsf{fwdLastGY}\ x\ t_x\ vx$ remains valid. The concurrent `write` may change the color of s to green, by invoking *forward* (Figure 5, line 5), but then *fx* becomes non-$\bot$, thus making $\mathsf{fwdLastGY}\ x\ t_x\ vx$ hold trivially.

In lines 15 and 17, `scan` reads from the forwarding pointers *fx* and *fy* and stores the obtained values into *ox* and *oy*, respectively. By Invariant 5, we know that if $ox \neq \bot$, there exists t'_x s.t. $t'_x \mapsto (x, ox) \in \chi$, and t'_x is the last green or yellow write event of χ_x. In case $ox = \bot$, we know from the $\mathsf{fwdLastGY}$ conjunct preceding the read from *fx*, that such last green or yellow event is exactly t_x. The consideration for *fy* is symmetric, giving us the assertion in line 18, where:

$$\mathsf{lastGYHist}\ p\ t\ v\ \mathrel{\widehat{=}}\ \mathsf{lastGY}\ p\ t \wedge t \mapsto (p, v) \in \chi$$

Next, line 19 merely names by rx the value of vx, if ox equals $\bot$, and similarly for ry in line 20, leading to line 21. Finally, on line 22, the method finishes by invoking $\langle$ *relink*(rx, ry); **return** $(rx, ry)\,\rangle$. Thus, it returns the selected snapshot (r_x, r_y) and relinks the events so that the Ω justifies the choice of snapshots.

We prove that the final state satisfies the postcondition in line 23, by using the main property of *relink* (Lemma 13). First, we pick $t = \mathsf{max}_\sigma(t'_x, t'_y)$. Then $r = \mathsf{eval}\ t\ \Omega\ \chi$ holds, by the following argument. By Lemma 13.1, rx is the value of the last green timestamp in χ_x. By Lemma 13.2, all the timestamps below t are green, thus rx is the value of the *last* timestamp in χ_x that is smaller or equal to t. By a symmetric argument, the same holds of ry. But then, the pair $r = (rx, ry)$ is the snapshot at t, *i.e.*, equals $\mathsf{eval}\ t\ \Omega\ \chi$.

The conjunct $t \in \mathsf{scanned}\,\Omega$ is proved as follows. Unfolding the definition of $\mathsf{scanned}$, we need to show $\Omega \downarrow t = \leqslant_\sigma \downarrow t$, and $\forall s \in \Omega \downarrow t.\, \kappa(s) = \mathsf{green}$. The first conjunct follows from Lemma 12. The second immediately follows from the first by Lemma 13.2.

To establish $\mathsf{dom}(h) \subseteq \Omega \downarrow t$, we proceed as follows. Let $s \in \mathsf{dom}(h)$. From line 21, we know $\kappa(s) = \mathsf{green}$. Because t'_x and t'_y are last green (by σ) or yellow events, by Invariant 4 it must be $s \leqslant_\sigma t'_x, t'_y$, and thus $s \leqslant_\sigma t$. However, we already showed that $\Omega \downarrow t = \leqslant_\sigma \downarrow t$. Thus, $s \ \Omega \ t$, finally establishing the postcondition.

8 Discussion

Comparison with linearizability, revisited As we argued in Section 3, our specifications for the snapshot methods directly capture that the method calls can be placed in a linear sequence, in a way that preserves the order of non-overlapping calls. This is precisely what linearizability achieves as well, but by technically different means. We here discuss some similarities and differences between our method and linearizability.

The first distinction is that linearizability is a property of a concurrent object, whereas our specifications are ascribed to individual methods, as customary in Hoare logic. This immediately enables us to use an of-the-shelf Hoare logic, such as FCSL, for specification.

Second, linearizability draws its power from the connection to contextual refinement [11]: one can substitute a potentially complex method A in a larger context, by a simpler method B, to which A linearizes. In our setting, such a property is enabled by a general substitution principle, which says that programs with the same spec can be interchanged in a larger context, without affecting the larger context's proof. Moreover, contextual refinement (and thus linearizability) is defined for general programs, without regard to their preconditions and postconditions. However, it is often the case that the refinement only holds if the substituted programs satisfy some Hoare logic spec. In this sense, our setting is more expressive, since the substitution principle is given relative to a Hoare logic spec.

Finally, while our specification of the snapshot methods are motivated by linearizability, there is no requirement—and hence no proof—that an FCSL specification implies linearizability. But this is a feature, rather than a bug. It enables us to specify and combine, in one and the same logic, programs that are linearizable, with those that are not. We refer to [38] for examples of how to specify and verify non-linearizable programs in FCSL.

Alternative snapshot implementations. FCSL's substitution principle can be exploited further in an orthogonal way: it allows us to re-use the specs for `write` and `scan` in Figure 4, ascribing them to a different concurrent snapshot algorithm. For that matter, we re-visit the previous verification in FCSL of the pair-snapshot algorithm [37]. We present only `scan` in Figure 9, as `write` is trivial.

```
scan () : $(A \times A)$ {
  $(cx, vx) \leftarrow \mathsf{read}(x)$;
  $(cy, \_) \leftarrow \mathsf{read}(y)$;
  $(\_, tx) \leftarrow \mathsf{read}(x)$;
  if $vx = tx$
  then return $(cx, cy)$
  else scan (); }
```

Figure 9 `scan` using versions.

In this example, the snapshot structure consists of pointers x and y storing tuples (c_x, v_x) and (c_y, v_y), respectively. c_x and c_y are the payload of x and y, whereas v_x and v_y are version numbers, internal to the structure. Writes to x and y increment the version number, while `scan` reads x, y and x again, in succession. Snapshot inconsistency is avoided by restarting if the two version numbers of x differ. In this paper's notation, the specification proved for `scan` in [37] reads:

$$\texttt{scan} : \{\chi_{\mathsf{s}} = \varnothing\} \ \{\exists t.\, \chi'_{\mathsf{s}} = \varnothing \wedge r = \mathsf{eval}\ t\ \chi' \wedge \mathsf{dom}(\chi) \subseteq \chi' \downarrow t\}$$

This spec is indeed very similar to the one of `scan` in Figure 4, but exhibits that the algorithm does not require dynamic modification to the event ordering. Thus, by defining Ω to be the natural ordering on timestamps in the global history χ (so that $\Omega' \downarrow t = \chi' \downarrow t$), and taking $\mathsf{scanned}\,\Omega$ to be the set of all timestamps in χ (so that $t \in \mathsf{scanned}\,\Omega$ is trivially true and can be added to the postcondition above), the above spec directly weakens into that of Figure 4. Since client proofs are developed in FCSL out of the specs, and not the code of programs, we can substitute different implementations of snapshot algorithms in clients, without disturbing the clients' proofs. This is akin to the property that programs that linearize to the same sequential code are interchangeable in clients.

Relation to Jayanti's original proof. Finally, we close this section by noting that our proof of Jayanti's algorithm seems very different from Jayanti's original proof. Jayanti relies on so-called *forwarding principles*, as a key property of the proof. For example, Jayanti's First Forwarding Principle says (in paraphrase) that if `scan` misses the value of a concurrent write through lines 10–11 of Figure 1, but the write terminates before the scanner goes through line 12 (the linearization point of `scan`), then the scanner will catch the value in the forwarding pointers through lines 13–14. Instead of forwarding principles, we rely on colors to algorithmically construct the status of each write event as it progresses through time, and express our assertions using formal logic. For example, though we did not use the First Forwarding Principle, we nevertheless can express a similar property, whose proof follows from Invariants introduced in Section 5:

▸ **Proposition 14.** If $S_s = \mathsf{Soff}\ t_{\mathsf{off}}$ and $S_x = S_y = \mathsf{True}$—i.e., the scanner is in lines 13–16 and it has unset S in line 12 at time t_{off}—then: $\forall t \in \chi.\ t \leqslant \tau(t) < t_{\mathsf{off}} \implies \kappa(t) = \mathsf{green}$.

9 Related work

Program logics for linearizability. The proof method for establishing linearizability of concurrent objects based on the notion of linearization points has been presented in the original paper by Herlihy and Wing [19]. The first Hoare-style logic, employing this method for compositional proofs of linearizability was introduced in Vafeiadis' PhD thesis [41, 42]. However, that logic, while being inspired by the combination of Rely-Guarantee reasoning and Concurrent Separation logic [43] with syntactic treatment of linearization points [42], did not connect reasoning about linearizability to the verification of client programs that make use of linearizable objects in a concurrent environment.

Both these shortcomings were addressed in more recent works on program logics for linearizability [26, 28], or, equivalently, *observational refinement* [11, 40]. These works provided semantically sound methodologies for verifying refinement of concurrent objects, by encoding atomic commands as resources (sometimes encoded via a more general notion of *tokens* [26]) directly into a Hoare logic. Moreover, the logics [28, 40] allowed one to give the objects standard Hoare-style specifications. However, in the works [28, 40], these two properties (*i.e.*, linearizability of a data structure and validity of its Hoare-style spec) are established separately, thus doubling the proving effort. That is, in those logics, provided a proof of linearizability for a concurrent data structure, manifested by a spec that suitably handles a *command-as-resource*, one should then devise a declarative specification that exhibits temporal and spatial aspects of executions (akin to our history-based specs from Figure 4), required for verifying the client code.

Importantly, in those logics, determining the linearization order of a procedure is tied with that procedure "running" the command-as-resource within its execution span. This makes it difficult to verify programs where the procedure terminates before the order is decided on, such as `write` operation in Jayanti's snapshot. The problem may be overcome by extending the scope of *prophecy variables* [2] or *speculations* beyond the body of the specified procedure. However, to the best of our knowledge, this has not been done yet.

Hoare-style specifications as an alternative to linearizability. A series of recent Hoare logics focus on specifying concurrent behavior *without* resorting to linearizability [8,24,37–39]. This paper continues the same line of thinking, building on [37], which explored patterns of assigning Hoare-style specifications with self/other auxiliary histories to concurrent objects, including *higher-order* ones (e.g., flat combiner [17]), and *non-linearizable* ones [38] in FCSL [31], but has not considered non-local, future-dependent linearization points, as required by Jayanti's algorithm.

Alternative logics, such as Iris [23,24] and iCAP [39], employ the idea of "ghost callbacks" [21], to identify precisely the point in code when the callback should be invoked. Such a program point essentially corresponds to a local linearization point. Similarly to the logical linearizability proofs, in the presence of future-dependent LPs, this method would require speculating about possible future execution of the callback, just as commented above, but that requires changes to these logics' metatheory, in order to support speculations, that have not been carried out yet.

The specification style of TaDA logic [8] is closer to ours in the sense that it employs *atomic tracking resources*, that are reminiscent of our history entries. However, the metatheory of TaDA does not support ownership transfer of the atomic tracking resources, which is crucial for verifying algorithms with non-local linearization points. As demonstrated by this paper and also previous works [37,38], history entries can be subject to ownership transfer, just like any other resources.

The key novelty of the current work with respect to previous results on Hoare logics with histories [3,12,13,16,28,37] is the idea of representing logical histories as auxiliary state, thus enabling constructive reasoning, by *relinking*, about dynamically changing linearization points. Since relinking is just a manipulation of otherwise standard auxiliary state, we were able to use FCSL *off the shelf*, with no extensions to its metatheory. Furthermore, we expect to be able to use FCSL's higher-order features to reason about higher-order (*i.e.*, parameterized by another data structure) snapshot-based constructions [34]. Related to our result, O'Hearn *et al.* have shown how to employ history-based reasoning and Hoare-style logic to *non-constructively* prove the existence of linearization points for concurrent objects out of the data structure invariants [32]; this result is known as *the Hindsight Lemma*. The reasoning principle presented in this paper generalizes that idea, since the Hindsight Lemma is only applicable to "pure" concurrent methods (*e.g.*, a concurrent set's `contains` [15]) that do not influence the position of other threads' linearization points. In contrast, our history relinking handles such cases, as showcased by Jayanti's construction, where the linearization point of `write` depends on the (future) outcome of `scan`.

Semantic proofs of linearizability. There has been a long line of research on establishing linearizability using forward-backwards simulations [6,7,35]. These proofs usually require a complex simulation argument and are not modular, because they require reasoning about the entire data structure implementation, with all its methods, as a monolithic STS.

Recent works [5,9,18] describe methods for establishing linearizability of sophisticated implementations (such as the Herlihy–Wing queue [19] or the time-stamped stack [9]) in a modular way, via *aspect-oriented* proofs. This methodology requires devising, for each class of objects (*e.g.*, queues or stacks), a set of specification-specific conditions, called *aspects*, characterizing the observed executions, and then showing that establishing such properties implies its linearizability. This approach circumvents the challenge of reasoning about future-dependent linearization points, at the expense of (a) developing suitable aspects for each new data structure class and proving the corresponding "aspect theorem", and (b) verifying the aspects for a specific implementation. Even though some of the aspects have been mechanized and proved adequate [9], currently, we are not aware of such aspects for snapshots.

Our approach is based on program logics and the use of STSs to describe the state-space of concurrent objects. Modular reasoning is achieved by means of separately proving properties of specific STS transitions, and then establishing specifications of programs, composed out of well-defined atomic commands, following the transitions, and respecting the STS invariants.

Proving linearizability using partial orders. Concurrently with us, Khyzha *et al.* [25] have developed a proof method for proving linearizability, which can handle certain class of data structures with similar future dependent behavior. The method works by introducing a partial order of events for the data structure as auxiliary state, which in turn defines the abstract histories used for satisfying the sequential specification of the data structure. Relations are added to this partial order at *commitment points* of the instrumented methods, which the verifier has to identify.

The ultimate goal of this method is to assert the linearizability of a concurrent data structure. As we have shown in Section 4, FCSL goes beyond as it provides a logical framework to carry out formal proofs about the correctness of a concurrent data structure and its clients.

The proof technique also tracks the ordering of events differently from ours. Where we keep a single witness for the current total ordering of events at all stages of execution, their technique requires keeping many witnesses. Their main theorem requires a proof that all linearizations of the abstract histories—*i.e.* all possible linear extensions of the partial order into a total order—satisfy the sequential specification of the data structure.

Through personal communication we learned that the technique cannot apply, for instance, to the verification of the *time-stamped* (TS) stack [9]. This is because a partial order does not suffice to characterize the abstract histories required to verify the data structure. In contrast, given the flexibility of FCSL in designing and reasoning with auxiliary state, we believe that our technique would not suffer such shortcomings.

10 Conclusions

The paper illustrates a new approach allowing one to specify that the execution history of a concurrent data structure can be seen as a *sequence of atomic events*. The approach is thus similar in its goals to linearizability, but is carried out exclusively using a separation-style logic to uniformly represent the state and time aspects of the data structure and its methods.

Reasoning about time using separation logic is very effective, as it naturally supports *dynamic and in-place updates* to the temporal ordering of events, much as separation logic supports dynamic and in-place updates of spatially linked lists. The need to modify the ordering of events frequently appears in linearizability proofs, and has been known to be tricky, especially when the order of a terminated event depends on the future. In our approach, the modification becomes a conceptually simple manipulation of auxiliary state of histories of colored timestamps.

We have carried out and mechanized our proof of Jayanti's algorithm [22] in FCSL, without needing any additions to the logic. Such development, together with the fact that

FCSL has previously been used to verify a number of non-trivial concurrent structures [36–38], gives us confidence that the approach will be applicable, with minor modifications, to other structures whose linearizations exhibit dynamic dependence on the future [9, 20, 29].

One modification that we envision will be in the design of the data type of timestamped histories. In the current paper, a history of the snapshot object needs to keep only the `write` events, but not the `scan` events. In contrast, in the case of stacks, a history would need to keep both events for push and pop operations. But in FCSL, histories are a *user-defined* concept, which is not hardwired into the semantics of the logic. Thus, the user can choose any particular notion of history, as long as it satisfies the properties of a Partial Commutative Monoid [27, 31]. Such a history can track pushes and pops, or any other auxiliary notion that may be required, such as, *e.g.*, specific ordering constraints on the events.

References

1 Concurrent data structures linked in time: Coq mechanization files. Available from `http://software.imdea.org/~germand/relink`.

2 Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *LICS*, pages 165–175. IEEE Computer Society, 1988.

3 Christian J. Bell, Andrew W. Appel, and David Walker. Concurrent separation logic for pipelined parallelization. In *SAS*, volume 6337 of *LNCS*, pages 151–166. Springer, 2010.

4 Andrea Cerone, Alexey Gotsman, and Hongseok Yang. Parameterised linearisability. In *ICALP*, volume 8573 of *LNCS*, pages 98–109. Springer, 2014.

5 Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. Aspect-oriented linearizability proofs. *LMCS*, 11(1), 2015.

6 Robert Colvin, Simon Doherty, and Lindsay Groves. Verifying concurrent data structures by simulation. *Electr. Notes Theor. Comput. Sci.*, 137(2):93–110, 2005.

7 Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a lazy concurrent list-based set algorithm. In *CAV*, volume 4144 of *LNCS*, pages 475–488. Springer, 2006.

8 Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A Logic for Time and Data Abstraction. In *ECOOP*, volume 8586 of *LNCS*, pages 207–231, 2014.

9 Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack. In *POPL*, pages 233–246. ACM, 2015.

10 Mike Dodds, Suresh Jagannathan, Matthew J. Parkinson, Kasper Svendsen, and Lars Birkedal. Verifying custom synchronization constructs using higher-order separation logic. *ACM Trans. Program. Lang. Syst.*, 38(2):4:1–4:72, 2016.

11 Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

12 Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*, volume 6269 of *LNCS*, pages 388–402. Springer, 2010.

13 Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. Verifying concurrent memory reclamation algorithms with grace. In *ESOP*, volume 7792, pages 249–269. Springer, 2013.

14 Alexey Gotsman and Hongseok Yang. Linearizability with ownership transfer. In *CONCUR*, volume 7454 of *LNCS*, pages 256–271. Springer, 2012.

15 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, volume 3974, pages 3–16. Springer, 2006.

16 Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. Modular verification of concurrency-aware linearizability. In *DISC*, volume 9363 of *LNCS*, pages 371–387. Springer, 2015.

17 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, pages 355–364. ACM, 2010.

18 Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR*, volume 8052 of *LNCS*, pages 242–256. Springer, 2013.

19 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

20 Moshe Hoffman, Ori Shalev, and Nir Shavit. The baskets queue. In *OPODIS*, LNCS, pages 401–414. Springer-Verlag, 2007.

21 Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282. ACM, 2011.

22 Prasad Jayanti. An optimal multi-writer snapshot algorithm. In *STOC*, pages 723–732. ACM, 2005.

23 Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *ICFP*, pages 256–269. ACM, 2016.

24 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650. ACM, 2015.

25 Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew J. Parkinson. Proving linearizability using partial orders. In *ESOP*, 2017. To Appear. Available at `https://sites.google.com/site/lazylinearizability/`.

26 Artem Khyzha, Alexey Gotsman, and Matthew J. Parkinson. A generic logic for proving linearizability. In *FM*, pages 426–443, 2016.

27 Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, pages 561–574. ACM, 2013.

28 Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470. ACM, 2013.

29 Adam Morrison and Yehuda Afek. Fast Concurrent Queues for x86 Processors. In *PPoPP*, pages 103–112, New York, NY, USA, 2013. ACM.

30 Aleksandar Nanevski. Separation logic and concurrency. Oregon programming languages summer school, 2016. `http://software.imdea.org/~aleks/oplss16/notes.pdf`.

31 Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, volume 8410 of *LNCS*, pages 290–310. Springer, 2014.

32 Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. Verifying linearizability with hindsight. In *PODC*, pages 85–94. ACM, 2010.

33 Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

34 Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *DISC*, volume 8205 of *LNCS*, pages 224–238. Springer, 2013.

35 Gerhard Schellhorn, Heike Wehrheim, and John Derrick. How to prove algorithms linearisable. In *CAV*, volume 7358 of *LNCS*, pages 243–259. Springer, 2012.

36 Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87. ACM, 2015.

37 Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *ESOP*, volume 9032, pages 333–358. Springer, 2015.

38 Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In *OOPSLA*, pages 92–110. ACM, 2016.

39 Kasper Svendsen and Lars Birkedal. Impredicative Concurrent Abstract Predicates. In *ESOP*, volume 8410 of *LNCS*, pages 149–168. Springer, 2014.

40 Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390. ACM, 2013.

41 Viktor Vafeiadis. *Modular fine-grained concurrency verification.* PhD thesis, University of Cambridge, 2007.

42 Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP*, pages 129–136. ACM, 2006.

43 Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, volume 4703, pages 256–271. Springer, 2007.

A A brief introduction to FCSL

A state of a resource in FCSL [31], such as that of snapshot data structure discussed in this paper, always consists of three distinct auxiliary variables that we name a_s, a_o and a_j. These stand for the abstract self state, other state, and shared (joint) state.

However, the user can pick the types of these variables based on the application. In this paper, we have chosen a_s and a_o to be histories, and have correspondingly named them χ_s and χ_o. On the other hand, a_j consists of all the other auxiliary components that we discussed, such as the variables χ_j, τ, κ, S_x, S_y, W_x and W_y. These variables become merely projections out of a_j.

It is essential that a_s and a_o have a common type, which moreover, exhibits the algebraic structure of a *partial commutative monoid* (PCM). A PCM requires a partial binary operation $\bullet$ which is commutative and associative, and has a unit. PCMs are important, as they give a generic way to define the inference rule for parallel composition.

$$\frac{e_1 : \{P_1\}\ A\ \{Q_1\}@C \quad e_2 : \{P_2\}\ B\ \{Q_2\}@C}{e_1 \parallel e_2 : \{P_1 \circledast P_2\}\ (A \times B)\ \{[r.1/r]Q_1 \circledast [r.2/r]Q_2\}@C}$$

Here, $\circledast$ is defined over state predicates P_1 and P_2 as follows.

$$(P_1 \circledast P_2)(a_\mathsf{s}, a_\mathsf{j}, a_\mathsf{o}) \iff \exists x_1\ x_2.\ a_\mathsf{s} = x_1 \bullet x_2 \wedge P_1(x_1, a_\mathsf{j}, x_2 \bullet a_\mathsf{o}) \wedge P_2(x_2, a_\mathsf{j}, x_1 \bullet a_\mathsf{o})$$

The inference rule, and the definition of $\circledast$, formalize the intuition that when a parent thread forks e_1 and e_2, then e_1 is part of the environment for e_2 and vice-versa. This is so because the *self* component a_s of the parent thread is split into x_1 and x_2; x_1 and x_2 become the *self* parts of e_1, and e_2 respectively, but x_2 is also added to the *other* component a_o of e_1, and dually, x_1 is added to the *other* component of e_2.

In this paper, the PCM we chose is that of histories, which are a PCM under the operation of disjoint union $\mathbin{\dot\cup}$, with the $\varnothing$ history as the unit. More common in separation logic is to use heaps, which, similarly to histories, form PCM under disjoint (heap) union and theempty heap, `empty`. In FCSL, these can be combined into a Cartesian product PCM, to enable reasoning about both space and time in the same system.

The frame rule is a special case of the parallel composition rule, obtained when e_2 is taken to be the idle thread.

$$\frac{e : \{P\}\ A\ \{Q\}@C}{e : \{P \circledast R\}\ A\ \{Q \circledast R\}@C}\ R \text{ is stable}$$

For the purpose of this paper, the rule is important because it allows us to generalize the specifications of `write` and `scan` from Figure 4. In that figure, both procedures start with the precondition that $\chi_{\mathsf{s}} = \varnothing$. But what do we do if the procedures are invoked by another one which has already completed a number of writes, and thus its χ_{s} is non-empty. By $\circledast$-ing with the frame predicate $R \mathrel{\widehat{=}} (\chi_{\mathsf{s}} = k)$, the frame rule allows us to generalize these specs into ones where the input history equals an arbitrary k:

$$\begin{aligned}
&\texttt{write}\ (p, v) : \ \{\chi_{\mathsf{s}} = k\} \\
&\qquad\qquad\quad \{\exists t.\, \chi'_{\mathsf{s}} = h \mathbin{\dot\cup} t \mapsto (p, v) \wedge \mathsf{dom}(\chi_{\mathsf{o}}) \cup \mathsf{scanned}\ \Omega \subseteq \Omega' \downarrow t\}@C \\
&\texttt{scan} : \ \{\chi_{\mathsf{s}} = k\} \\
&\qquad \{r.\, \exists t.\, \chi'_{\mathsf{s}} = k \wedge r = \mathsf{eval}\ t\, \Omega'\, \chi' \wedge \mathsf{dom}(\chi) \subseteq \Omega' \downarrow t \wedge t \in \mathsf{scanned}\, \Omega'\}@C
\end{aligned}$$

These two *large-footprint* instances of the rules for `scan` and `write` are those used in the proof of our clients in Section 4. For further details on FCSL, its semantics and implementation, we refer the reader to [31].

B Implementation and Correctness of *relink*

In Section 6, we described briefly the implementation of *relink*, without giving much details on the auxiliary helper functions *inspect* and *push*. We give here their definitions, together with some associated properties:

▸ **Definition 15** (inspect)**.** Given two timestamps t_x, t_y then *inspect* $t_x\, t_y\, \sigma\, \kappa$ is defined as follows:

$$inspect\ t_x\, t_y\ \kappa \ \mathrel{\widehat{=}} \begin{cases} \mathsf{Yes}\ x\ t_z & \text{if } t_x <_\sigma t_y,\ t_x = \mathsf{last\_green}_\sigma\ \chi_x, \\ & t_z = \mathsf{yellow\_timestamp}_\sigma\ \chi_x, \text{ and } t_z <_\sigma t_y \\ \mathsf{Yes}\ y\ t_z & \text{if } t_y <_\sigma t_x,\ t_y = \mathsf{last\_green}_\sigma\ \chi_y, \\ & t_z = \mathsf{yellow\_timestamp}_\sigma\ \chi_y, \text{ and } t_z <_\sigma t_x \\ \mathsf{No} & \text{otherwise} \end{cases}$$

▸ **Definition 16** (push)**.** *push* is a surgery operation defined on σ as follows:

Let $\sigma = \sigma_{<_i} \mathbin{+\!\!+} i \mathbin{+\!\!+} \sigma_{i..j} \mathbin{+\!\!+} j \mathbin{+\!\!+} \sigma_{>_j}$, then *push* $i\ j\ \sigma = \sigma_{<_i} \mathbin{+\!\!+} \sigma_{i..j} \mathbin{+\!\!+} j \mathbin{+\!\!+} i \mathbin{+\!\!+} \sigma_{>_j}$

The definition of *inspect* works under the assumption that t_x and t_y are, respectively, the last green or yellow timestamp in χ_x and χ_y. This latter fact is recovered in the definition of *relink* in Figure 6 and reinforced in line 21 in the proof of `scan` in Figure 8. When *inspect* returns $\mathsf{Yes}\ p\, t_z$, σ' is computed by pushing some i timestamp past another timestamp j in σ. The definition of *push* above shows that this operation is an algebraic manipulation on sequences. In fact, we implement it using standard *surgery* operations on lists: $\mathbin{+\!\!+}$, take, *etc.*

In Section 6, we have mentioned that the correctness aspect of auxiliary code involves proving that the code preserves the auxiliary state invariants from Section 5. For example, the correctness proof of *relink*, relies on the following helper lemmas. The first lemma asserts that *inspect* correctly determines the "offending" timestamp; the second and the third lemma assert that *push* modifies σ in a way that allows us to prove (in Section 7), that the pair (r_x, r_y) a valid snapshot.

▸ **Lemma 17** (Correctness of *inspect*)**.** *If t_x, t_y are timestamps for write events of r_x, r_y, then inspect $t_x\ t_y\ \sigma\ \kappa$ correctly determines that (r_x, r_y) is a valid snapshot under ordering $<_\sigma$ and colors κ, or otherwise returns the "offending" timestamp. More formally, if $S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}}, S_x = \mathsf{True}, S_y = \mathsf{True}$, and for each $p \in \{x, y\}$, $t_p \mapsto (p, r_p) \in \chi$ and $\mathsf{lastGY}\ p\, t_p$, the following are exhaustive possibilities.*

1. *If* $t_x <_\sigma t_y$ *and* $\kappa(t_x) = \mathsf{yellow}$*, then inspect* $t_x\ t_y\ \sigma\ \kappa = \mathsf{No}$*. Symmetrically for* $t_y <_\sigma t_x$*.*
2. *If* $t_x <_\sigma t_y$*,* $t_x =$ *last_green* χ_x*, and* $\forall s \in \chi_x.\ t_x <_\sigma s \implies t_y <_\sigma s$*, then inspect* $t_x\, t_y\, \sigma\, \kappa = \mathsf{No}$*. Symmetrically for* $t_y <_\sigma t_x$*.*
3. *If* $t_x <_\sigma t_y$*,* $t_x =$ *last_green* χ_x*,* $s \in \chi_x$*, and* $t_x <_\sigma s <_\sigma t_y$*, it follows that inspect* $t_x\, t_y \sigma \kappa = \mathsf{Yes}\, x\, s$ *and* $\kappa(s) = \mathsf{yellow}$*. Symmetrically for* $t_y <_\sigma t_x$*.*

▸ **Lemma 18** (Push Mono)**.** *Given elements* a, b, i, j*, all in* σ*, and* $\sigma' =$ *push* $i\, j\, \sigma$*, then:*
1. *If* $a <_\sigma i$ *then* $a <_\sigma b \implies a <'_\sigma b$*.*
2. *If* $j <_\sigma b$ *then* $a <_\sigma b \implies a <'_\sigma b$*.*
3. *If* $a \neq i$ *then* $a <_\sigma b \implies a <'_\sigma b$

▸ **Lemma 19** (Correctness of *push*)**.** *Given* $S_s = \mathsf{S_{Off}}\ t_{\mathsf{off}}, S_x = S_y = \mathsf{True}$*, and for* $p \in \{x, y\}$*, we have* $t_p \mapsto (p, r_p) \in \chi$*,* $\mathsf{lastGY}\ p\, t_p$*, and inspect* $t_x\ t_y\ \sigma\ \kappa = \mathsf{Yes}\ p\ t_s$*. If we name* $t_z \in \{t_x, t_y\}$*, with* $p \neq z$*, and* $\sigma' =$ *push* $t_s\ t_z\ \sigma$*, then:*
1. *relink satisfies the 2-state invariants from Invariant 1.*
2. $\chi', \sigma', \tau', \kappa'$ *satisfies all the resource invariants from Section 5,* i.e. *Invariants 3–10.*

In our mechanization, these three lemmas allow us to prove Lemma 13, *relink*'s main property.