Visibility and Separability for a Declarative Linearizability Proof of the Timestamped Stack

Jesús Domínguez
IMDEA Software Institute, Spain
Universidad Politécnica de Madrid, Spain

Aleksandar Nanevski
IMDEA Software Institute, Spain

Abstract

Linearizability is a standard correctness criterion for concurrent algorithms, typically proved by establishing the algorithms’ linearization points (LP). However, LPs often hinder abstraction, and for some algorithms such as the timestamped stack, it is unclear how to even identify their LPs. In this paper, we show how to develop declarative proofs of linearizability by foregoing LPs and instead employing axiomatization of so-called visibility relations. While visibility relations have been considered before for the timestamped stack, our study is the first to show how to derive the axiomatization systematically and intuitively from the sequential specification of the stack. In addition to the visibility relation, a novel separability relation emerges to generalize real-time precedence of procedure invocation. The visibility and separability relations have natural definitions for the timestamped stack, and enable a novel proof that reduces the algorithm to a simplified form where the timestamps are generated atomically.

1 Introduction

A concurrent data structure is linearizable [11] if in every concurrent execution history of the structure’s exportable methods, the method invocations can be ordered linearly just by permuting overlapping invocations, so that the obtained history is sequentially sound; that is, executing the methods sequentially in the linear order produces the same outputs that the methods had in the concurrent history. In other words, every concurrent history is equivalent to a sequential one where methods execute without interference, i.e., atomically.

While linearizability is a standard correctness criterion, proving that sophisticated data structures are linearizable is far from trivial. The most common approach is to first describe the linearization points (LPs) of the methods that the data structure exports. Given an execution of a method (henceforth, event), its LP is the moment at which the event’s effect can be considered to have occurred abstractly, in the sense that the linearization order of the events is determined by the real-time order of the chosen LPs. LPs are described operationally by indicating the line in the code together with a run-time condition under which the line applies. The proof then proceeds by a simulation argument to show that the effect of the invocation abstractly occurs at the declared line.

While LPs lead to a complete proof method [16], the operational nature of the LP

© Jesús Domínguez and Aleksandar Nanevski; licensed under Creative Commons License CC-BY 4.0
34th International Conference on Concurrency Theory (CONCUR 2023).
Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 30; pp. 30:1–30:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
description leads to very low-level proofs. Sometimes, it may even be unclear how to describe the position of the LPs in the first place. An alternative, more declarative approach, that offers higher levels of abstraction, has been proposed by Henzinger et al. [10]. It advocates foregoing LPs in favor of axiomatizing how the events of the structure depend on each other. Such dependence relation has since been termed visibility relation in the literature [17], and has been used to axiomatize concurrent queues [10], stacks [3, 7], and snapshot algorithms [14]. In these cases, the higher abstraction capabilities of visibility relations (compared to LPs) enabled that linearizability proofs of different implementations of a data structure can share significant proof components, or that a linearizability proof can be developed in the first place where an LP-based proof did not exist. Nevertheless, despite these recent successes, developing visibility-based proofs remains an undeveloped area, with every proof approaching the axiomatization in its own manner, without any specific systematization.

This paper advances the visibility approach by proposing that the axiomatization of concurrent structures should rely on a separability relation between events, in addition to the visibility relation. Separability relation partially characterizes when two events are abstractly non-overlapping, with one event logically preceding the other. Thus, it is the abstract counterpart to the “returns-before” relation, which is standard in the literature, and holds between two events if, in real time, the first event terminates before the second begins.

We employ the visibility and separability relations in tandem to derive a new axiomatization and linearizability proof for the concurrent structure of the timestamped stack, initially designed and proved linearizable by Dodds et al. [3] and Haas [7]. Since its inception, the timestamped stack has achieved some notoriety for the difficulty of its linearizability proof, as it has so far resisted an operational description of its LPs, and simulation-based proof attempts. For example, Khyzha et al. [12] verified the timestamped queue by a simulation-based approach, but did not scale to the stack. Bouajjani et al. [1] employed forward simulation on a simplified variant of the stack where timestamps are allocated atomically, but did not attempt the general variant, where the timestamp allocation is a more complex non-atomic operation that produces behaviors not observed in the atomic case. The original proof by Dodds et al. is a large case analysis that mixes visibility relations with approximate LP descriptions, and then adapts and corrects both as the proof advances. However, the axioms and the definitions of the visibility relations have been justified only technically, and have remained unconnected to the intuition behind the structure’s design.

By axiomatizing both separability and visibility relations, we derive the following contributions: (1) We obtain a linearizability proof that elides LPs, and is thus more declarative than the proof of Dodds et al.; (2) The proof’s declarative nature allows us to first consider the simpler variant of the algorithm with atomic timestamp allocation, and then show that the general variant reduces to the atomic case. The staged proof is more intuitive than if we attempted the general case directly, which is what Dodds et al. do.; (3) Our contribution goes beyond a new proof for the timestamped stack, as it suggests a systematic way to axiomatize concurrent data structures in the visibility style. More specifically, we show that the visibility relation naturally emerges when one transforms an obvious state-based sequential axiomatization of stacks to the concurrent setting with histories. In the process, the separability relation also naturally emerges, because one is immediately forced to generalize the returns-before relation. So obtained axioms identify the abstractions that are essential for understanding the algorithm, and strongly guide the remaining proof. Finally, our approach to axiomatization applies to other concurrent algorithms as well, and we comment in Section 5 how we did so for RDCSS and MCAS of Harris et al. [8] and some other structures. Of course, the generality of the approach remains to be evaluated on a wider set of examples.
The timestamped stack (TS-stack) keeps an array of pools, indexed by thread IDs; one pool for each thread. A pool is a linked list of nodes. The array index identifying the pool (line 1 in Figure 1) stores the head node of the pool list, and each node (lines 22-23) stores a value \( \text{val} \), timestamp \( \text{stamp} \), the next node in the list, a boolean \( \text{taken} \) indicating if the value has been taken by some pop, and a unique identifier \( \text{id} \) for the node.\(^1\) Each thread can only insert values in its own pool by allocating a node at the head of the list. A value is logically removed from the pool once its \( \text{taken} \) flag is set to true.\(^2\)

The \textit{push} procedure inserts a new node containing the pushed value \( v \) into the pool of the executing thread with thread id \( \text{TID} \). More specifically, in line 6, \textit{push} allocates a new node with the value \( v \), infinite timestamp, \( \text{next} \) pointing to the current head of the pool, taken flag set to false, and fresh unique identifier, where \( \text{ID}++ \) denotes an atomic fetch and increment on the global counter \( \text{ID} \). Then, the new node is set as the new head of the \( \text{TID} \) pool (line 7), a new timestamp is generated (line 8) and assigned to the node (line 9) as a replacement for the original infinity timestamp. We will discuss infinity timestamps and the \textit{newTimestamp} procedure further below.

The \textit{pop} procedure traverses the pools (loop at line 30), searching for anuntaken node with a maximal timestamp in the partial order \( <_T \), updating the current maximum in the variable \( \text{maxT} \) (lines 35-36). Once a maximal node is found, \textit{pop} attempts to remove it by CAS-ing on its \( \text{taken} \) flag at line 38. The \textit{pop} procedure restarts (loop at line 27) if it was not able to take a maximal node at line 38.

The role of \( <_T \) is to endow TS-stack with a LIFO discipline whereby an element with a larger timestamp (i.e. the more recently pushed element), is popped first. In the concurrent setting, however, the meaning of “more recent” is not as straightforward as in the sequential setting, as the definition of linearizability allows that overlapping operations can be linearized.

---

1. Unique identifiers \( \text{id} \) are ghost code (gray color in Figure 1), introduced solely for use in proofs.
2. For presentation purposes, we simplified the original algorithm, but treat a more general form in Appendix B.2 [5]. The two versions exhibit the same challenges, and use the same axiomatization and definitions of visibility and separability. The differences between them are discussed in Section 5.
in either order. In particular, if two invocations of push overlapped, they can actually be popped in either order. To reflect this property of linearizability, the order $<_T$ is partial as opposed to total. However, to be sequentially sound, it is of essence that if two pushes did not overlap, then the more recent push is indeed popped first.\(^3\) This is why the implementation of newTimestamp should satisfy the property that two non-overlapping calls to newTimestamp produce timestamps that actually are ordered by $<_T$.

There are several ways in which one can implement newTimestamp to satisfy this property, and Figure 1 shows the particularly efficient variant proposed by Dodds et al. [3]. We will return to this variant promptly. However, for purposes of understanding and proving the algorithm linearizable, one may consider a simpler version whereby timestamps are integers, and newTimestamp is implemented to keep a global counter that is atomically fetch-and-incremented on each call, returning the current count as the fresh timestamp. Such an atomic implementation results in $<_T$ that is actually a total order, and much simpler to analyze than the efficient variant in Figure 1. We will use the atomic implementation as a stepping stone in our proof; we will prove it linearizable first, and then show that the linearizability argument for the efficient variant reduces to the atomic case.

The reason to consider a non-atomic implementation at all is that the atomic one suffers from a performance issue that threads contend on the global timestamp counter. The efficient variant from Figure 1 improves on this by introducing interval timestamps of the form $[a, b]$ for integers $a \leq b$, where $[a, b] <_T [c, d]$ holds if $b < c$ in the standard integer order. Obviously, so defined $<_T$ is only a partial order, as it does not order every two interval timestamps. Nevertheless, it still suffices for linearizability, because if two push events are assigned overlapping interval timestamps, such events must overlap as well, and thus do not constrain the order in which they are popped in a linearization.

The newTimestamp from Figure 1 still keeps a global counter $TS$, as the atomic variant would, but it does not always synchronize accesses to it. In particular, $TS$ is first read twice into $ts_1$ and $ts_2$ (lines 12 and 14, respectively). In the common case when some thread interfered on $TS$ (i.e., $ts_1 \neq ts_2$), the method generates an interval with endpoint $ts_2 - 1$, and terminates without having performed any synchronization. Some synchronization is required only when no interference is detected (i.e., $ts_1 = ts_2$). In that case, newTimestamp CAS-es over $TS$ (line 17), to atomically increment $TS$. As CAS is an expensive operation, invoking pause() in line 13 increases the probability of interference, and thus decreases the need for CAS. If the CAS succeeds, an interval with endpoint $ts_1$ is returned. If the CAS fails, some other thread increased $TS$, and the method returns an endpoint $TS - 1$. In all cases, when newTimestamp terminates, $TS$ has been increased either by the executing thread or by another thread, and the generated interval’s endpoint is strictly smaller than the current value of $TS$. Thus, a subsequent non-overlapping invocation of newTimestamp will produce an interval that is strictly larger in $<_T$. This ensures that two sequentially non-overlapping pushes generate non-overlapping interval timestamps.

Note that newTimestamp could return the same interval timestamp for two different overlapping invocations. For example, with initial $TS = 0$, a thread $T_1$, after reading $TS$ the first time at line 12 (returning 0), waits at line 13 while another thread $T_2$ fully executes newTimestamp, meaning that $T_2$ increased $TS$ at line 17 and returned timestamp $[0, 0]$. When $T_1$ resumes, it again reads $TS$ at line 14 (returning 1), and so returns $[0, 0]$.

Finally, $<_T$ is formally augmented with infinite timestamps $-\infty$ and $\infty$, so that $-\infty <_T t <_T \infty$ for any timestamp $t$ generated by newTimestamp. This enables pop to start its search

\(^3\) Further assuming that the pops also did not overlap among themselves or with the pushes.
(A1) Non-empty pop
\[ v :: S \xrightarrow{\text{pop}(v)} S \]

(A2) Empty pop
\[ S \xrightarrow{\text{pop}() \ (\text{EMPTY})} [] \]

(A3) push
\[ S \xrightarrow{\text{push}(v) \ (tt)} v :: S \]

(a) State-based sequential specification.

(B1) LIFO
\[ u_1 < o_1 \land u_1 \sqsubset u_2 \sqsubset o_1 \implies \exists o_2. u_2 < o_2 \land o_2 \sqsubset o_1 \]

(B2) Pop uniqueness
\[ u < o_1 \land u < o_2 \implies o_1 = o_2 \]

(B3) Dependencies occur in the past
\[ u < o \implies u \sqsubset o \]

(B4.1) Non-empty pop
\[ o = \text{pop}() \ (v) \land v \neq \text{EMPTY} \implies \exists u. u < o \land v = u.\text{in} \]

(B4.2) Empty pop
\[ o_1 = \text{pop}() \ (\text{EMPTY}) \implies \forall u. u \sqsubset o_1 \implies \exists o_2. u < o_2 \land o_2 \sqsubset o_1 \]

(B4.3) push
\[ u = \text{push}() \ (tt) \implies v = tt \]

(b) History-based sequential specification. Relation \( < : \text{Ev} \times \text{Ev} \) is abstract, and \( u.\text{in} \) is event \( u \)'s input.

Figure 2 State-based and history-based sequential specifications for stacks. Variables \( u, o, \) and their indexed variants, range over pushes and pops, respectively.

with minimum timestamp \(-\infty\) (line 28). Similarly, \( \text{push} \) can assign maximum timestamp \( \infty \) to a fresh node (line 6) before assigning it a finite timestamp; an intervening \( \text{pop} \) could take such a fresh node immediately, as the node is the most recent.

3 Axiomatizing Visibility and Separability

3.1 Sequential History Specifications and Visibility Relations

Following Henzinger et al. [10], we start the development of visibility relations by introducing history-based specifications for our data structure. History-based specifications describe relationships between the data structure’s procedures in an execution history. They are significantly different from the perhaps more customary state-based specifications that describe the actions of a procedure in terms of input and output state. However, history-based specifications scale better to the concurrent setting, which is why concurrent consistency criteria such as linearizability are invariably defined in terms of execution histories.

In this section we focus on sequential histories in order to introduce the idea of visibility relation in a simple way, before generalizing to concurrent histories in Section 3.2. A sequential history is a sequence of the form \( [\text{proc}(\text{in}_1)\ (\text{out}_1), \ldots, \text{proc}(\text{in}_n)\ (\text{out}_n)] \), where \( \text{proc}(\text{in}_i)\ (\text{out}_i) \) means that \( \text{proc}(\text{in}_i) \) executed atomically and produced output \( \text{out}_i \). We term event each element in a sequential history \( h \), and \( \text{Ev} \) denotes the set of all events in \( h \).

Figure 2 illustrates the distinction between sequential state-based and history-based specifications for stacks. For the state-based specification in Figure 2a, let us denote by \( S \xrightarrow{\text{proc}(\text{in})\ (\text{out})} S' \) the statement that event \( \text{proc} \) with input \( \text{in} \) executes atomically on stack \( S \), produces output \( \text{out} \) and modifies the stack into \( S' \). Axiom \( A_1 \) says that a \( \text{pop} \) removes the top element \( v \) from a non-empty stack and returns \( v \). Axiom \( A_2 \) says that \( \text{pop} \) returns \( \text{EMPTY} \) when the stack is empty, leaving the stack unchanged. Axiom \( A_3 \) says \( \text{push}(v) \) inserts \( v \) into the stack as the new top element, returning the trivial value \( tt \).

Figure 2b shows the history-based sequential specification for stacks. The specification utilizes the visibility relation \( < \) to capture a push-pop causal dependence between events. In
Visibility and Separability for a Declarative Proof of the Timestamped Stack

particular, \( u \prec o \) means that “event \( o \) pops a value that event \( u \) pushed onto the stack”. We usually say that \( u \) is visible to \( o \), or that \( o \) observes \( u \). Under this interpretation, axioms \( B_1, ..., B_{4.3} \) state the following expected properties.

Axiom \( B_1 \) (LIFO) states that more recent pushes are popped first. More specifically, if \( o_1 \) observes \( u_1 \) (i.e., \( u_1 \prec o_1 \)) and \( u_2 \) is a later push executing between \( u_1 \) and \( o_1 \) (i.e., \( u_1 \sqsubset u_2 \sqsubset o_1 \)), then \( u_2 \) must be popped before \( o_1 \) pops \( u_1 \), otherwise the value pushed by \( u_1 \) would not be at the top of the stack for \( o_1 \) to take. Relation \( \sqsubset \) is the returns-before relation (with \( \sqsubseteq \) its reflexive closure), where \( x \sqsubseteq y \) means that \( x \) terminated before \( y \) started. Note that \( \sqsubset \) is a total order on events, as in a sequential execution, different events cannot overlap.

Axiom \( B_2 \) (Pop uniqueness) says that a push is observed by at most one pop.

Axioms \( B_{4.1} \) (Non-empty pop), \( B_{4.2} \) (Empty pop), and \( B_{4.3} \) essentially are the counterparts of the state-based sequential axioms \( A_1-A_3 \), respectively, as we show next.

Axiom \( B_{4.1} \) says that a \( \text{pop}() \) event \( o \) observes a push \( u \) that pushed \( v \). This axiom, along with \( B_1-B_4 \), ensures that \( o \) relates to \( u \) as in the following diagram.

\[
S \xrightarrow{u} v :: S \xrightarrow{o_1} v :: S \xrightarrow{u_k} \ldots o_j \xrightarrow{o_i} v :: S \xrightarrow{o = \text{pop}() (v)} S
\]

In particular: (i) \( u \) executes before \( o \) (by axiom \( B_3 \), because \( u \prec o \)), (ii) every push between \( u \) and \( o \) is popped before \( o \) (by axiom \( B_1 \)), and each push is popped exactly once (by axiom \( B_2 \)). Thus, once \( o \) executes, the value \( v \) pushed by the observed \( u \), is actually the most recent unpopped value, i.e., it is on the top of the stack. Subsequent pops cannot observe this value anymore either (again by axioms \( B_1 \) and \( B_2 \)), thus the stack is modified from \( v :: S \) to \( S \). This explains that \( B_{4.1} \) is essentially a history-based version of \( A_1 \).

Similarly, axiom \( B_{4.2} \) states that if a \( \text{pop}() \) (EMPTY) event \( o \) occurs, then every push before \( o \) must have been popped before \( o \) as this ensures that the stack is empty when \( o \) is reached. Hence, the axiom is counterpart to \( A_2 \).

Finally, axiom \( B_{4.3} \) says that the output of a push event is the trivial value \( tt \). The axiom imposes no conditions on the stack, as a value can always be pushed. In this, \( B_{4.3} \) is the counterpart to \( A_3 \) which also imposes no conditions on the input stack, and posits that push’s output value is trivial. However, unlike \( A_3 \), \( B_{4.3} \) does not directly says that the value is pushed on the top of the stack, as that aspect is captured by the relationships between pushes and pops described by \( B_{4.1} \).

### 3.2 Concurrent Specifications and Separability Relations

Concurrent execution histories do not satisfy the sequential axioms in Figure 2b for two related reasons. First, concurrent events can overlap in real time. As a consequence, the axioms \( B_1 \) (LIFO), \( B_3 \) (Dependencies occur in the past), and \( B_{4.2} \) (Empty pop) are too restrictive, as they force events to be non-overlapping (i.e., disjoint in time) due to the use of the returns-before relation \( \sqsubset \). Second, events can no longer be treated as atomic; thus event’s

---

4 Our paper will make heavy use of several different relations. To help the reader keep track of them, we denote the relations by symbols that graphically associate to the relation’s meaning. For example, we use \( \prec \) for the visibility relation, because the symbol graphically resembles an eye.
These rules have the following justification. Let us suppose we have a formula \( b \equiv a \sqcap b \rightarrow c \sqcap d \) in some sequential axiom. In the sequential case, \( \sqcap \) is a total order, which means that \( \phi \) is equivalent to \( b \sqsubseteq a \lor c \sqsubseteq d \). After directly replacing \( \sqcap \) for \( \kappa \), we obtain \( b \nless a \lor c \nless d \), which is further equivalent to \( \psi \equiv b \nless a \rightarrow c \nless d \). Comparing \( \phi \) and \( \psi \), we see that \( \psi \)'s condition is flipped, replaced, and negated, while its conclusion is only replaced. An important aspect of our procedure is that negative occurrences of \( \kappa \) in \( \psi \) are themselves negated. Thus, intuitively, \( \psi \) as a whole remains positive with respect to \( \kappa \).

5 The symbol \( \kappa \) twists \( \sqcap \), suggesting that \( \kappa \) relaxes (i.e., is a twist on) returns-before relation \( \sqcap \).
Positive formulas remain true under extensions of $\ll$, which is crucial, as the linearizability proof will involve extending $\ll$ until reaching a total order.

Axiom $C_2$ is unchanged compared to $B_2$.

Axiom $C_3$ is obtained from $B_3$ as follows. In the sequential specification, $\ll$ was the only relation encoding dependences between events, but now we have two relations encoding dependences, $\ll$ and $\ll$. To collect them, we define a new relation $\ll \equiv \ll \cup \ll$ which we call constraint relation. We can consider modifying Axiom $B_3$ into $x \ll y \Rightarrow x \ll c^t y$ to say that any dependence $x$ of $y$ must terminate before $y$ starts. However, such a modification of $B_3$ is too stringent, as it does not allow $x$ to overlap with $y$. Instead, we relax the conclusion to say that an event cannot depend on itself or events from the future, i.e., $x \ll y \Rightarrow y \not\ll x$. Finally, we get axiom $C_3$ by replacing $\ll$ with its transitive closure $\ll^+$ to account for indirect dependences of $y$; e.g., in $x_1 \ll x_2 \ll y$, event $x_1$ is an indirect dependence of $y$. Hence, $C_3$ reads “any direct or indirect dependence does not execute in the future, and events do not depend on themselves”.

To understand Axiom $C_4$, we need to consider the set $T$ of all terminated events and its closure under the constraint relation $\overline{T} \equiv \{ e \in Ev \mid \exists t \in T. e \ll^* t \}$. As usual, $\ll^*$ is the reflexive-transitive closure of $\ll$. The reason for considering this set is that the variable $x$ over which the axiom implicitly quantifies ranges over $\overline{T}$.

It is standard in linearizability that the linearization order contains all the terminated events, plus selected unterminated events with fictitious, but suitable, outputs. The selected unterminated events are typically those that executed their effect, which then influenced others, and must thus be included for sequential soundness. The set $\overline{T}$ precisely determines the events to be included by saturating the set of terminated events $T$ under $\ll$.

Axiom $C_4$ then codifies when an output $v$ is suitable for an event $x$ by means of the postcondition predicate $Q_{x,v}$. In particular, $C_4$ says that $v$ exists such that $Q_{x,v}$. If $x \in \overline{T}$ is unterminated, we use that $v$ as the fictitious output. If $x$ is terminated ($x \in T$), then $v$ must be $x$’s actual output. The postcondition predicate $Q_{x,v}$ describes how $x$ and $v$ relate in the case of stacks. It is obtained by coalescing the axioms $B_{4,1}$, $B_{4,2}$ and $B_{4,3}$, which themselves describe the outputs of stack events in the sequential setting, and which we first modify according to the systematic transformation outlined above.

We henceforth call the axioms in Figure 3, visibility-style axioms. These axioms imply linearizability of any stack implementation satisfying them.

**Theorem 3.1.** Let $D$ be an arbitrary implementation of a concurrent stack. If there are relations $\ll$ and $\ll$ definable using $D$ such that the visibility-style axioms hold, then $D$ is linearizable.

The proof starts with the relation $\ll \equiv (\ll \cup \ll)^*$, i.e, the transitive closure of the union of $\ll$ and $\ll$. Then, it shows that $\ll$ is a partial order that can be extended to a sequentially sound total order $\leq$ by using the visibility-style axioms. Since $\leq$ contains $\ll$, this means that relations $\ll$ and $\ll$ define ordering constraints that linearization respects.

### 4 Visibility and Separability for the TS-stack

By Theorem 3.1, to prove linearizability for the TS-stack, it suffices to define the relations $\ll$ and $\ll$ and show that they satisfy the axioms from Figure 3. We carry out this proof in two stages: In Section 4.2 we prove linearizability when the newTimestamp procedure

---

6 The symbol $\ll$ is like an eye with no iris; thus, “blinder” than $\ll$, reflecting that $\ll$ is a superset of $\ll$. 

---
is implemented by an atomic fetch-and-increment operation on a global counter $TS$, as discussed in Section 2. In Section 4.3 we show how the general case of interval timestamps reduces to the atomic case. The lifting exploits that the difference between the atomic and interval cases is only in the implementation of $newTimestamp$.

For simplicity, in both cases we explicitly exclude elimination pairs from the discussion. An elimination pair consists of a push and an overlapping pop event that takes the value pushed. The elision allows the discussion to only consider pushes with finite timestamps. Indeed, every push is first assigned an infinite timestamp (line 6 in Figure 1), which is then refined into a finite one in line 9. If a push $u$, having not yet reached line 9, is taken by some pop $o$, then $u$ and $o$ overlap, and hence form an elimination pair.\footnote{Eliding elimination pairs when dealing with stacks is justified because such pairs can be linearized simply as a push that is immediately followed by a pop. The idea was originated by Hendler et al. [9] and was also employed in Haas’ PhD dissertation [7], though with a different motivation from us and with a different soundness proof. For example, to prove the elimination sound, Haas shows how elimination pairs could be put back into the histories from which they have been removed. In contrast, we define visibility and separability relations that exclude elimination pairs, and show in Appendix B.1 [5], how to extend the relations iteratively, one elimination pair at a time. The extension adds some bulk, but does not change the structure of the proof that we illustrate in this section.}

Also, in both cases, we utilize the abstraction we call spans, to define the visibility and separability relations. A span of an event is the interval in which the event accesses the shared state of the stack. We could trivially take the span to be the whole interval of the event, but in the case of TS-stack we can tighten it as discussed below. In this sense, a span is a generalization of LPs; being an interval, rather than a single point, it approximates where the LP of an event lies, but allows for some uncertainty as to the LPs exact position.

The span of the push procedure starts when the new node is linked as the first node of the pool (line 7), as this is the moment when the new node becomes available for other events to see. The span ends when a finite timestamp is assigned to the new node (line 9). Notice how the span encompasses all the commands of push that change the pool or the new node.

The span of the pop procedure starts at the infinity stamp assignment (line 28) of the last iteration of the pools scan. The span ends at the successful CAS at line 38 which takes the node for the pop to return. Again, the span covers all the commands of pop that change the pools or the taken node. These are all included in the last iteration of the pools scan, because in all the prior iterations, the CAS modifying the pools must have failed.

We formalize spans as pairs of rep events $(a, b)$, where $a$ and $b$ are the initial and final rep event in the span, respectively. We denote by $start(b)$ and $end(b)$ the standard projection functions for span $b$. Rep events are generated by the invocation of a code line inside a procedure. For example, invoking line 7 in Figure 1 produces a rep event. The set of all rep events in an execution history is denoted as $Rep$. The distinction between events (Ev) and rep events is standard in linearizability [11]. We denote by $<$ the real-time order between rep events. We consider only fully-formed spans; for example, if pop has not executed its successful CAS, then it has no span.

We also extend our notion of timestamp into abstract timestamp. An abstract timestamp is a pair $(i, t)$, where $i$ is the (ghost) id of a node, and $t$ is a (plain) timestamp. The extension is motivated by the observation explained in Section 2 that two pushes may actually generate the same (plain) timestamp. By attaching the node id $i$ to the timestamp $t$, we differentiate such cases. We use “timestamp” to refer to abstract timestamps or plain timestamps when the adjective can be inferred from the context.

We also utilize the following notation.
Visibility and Separability for a Declarative Proof of the Timestamped Stack

- Given event $e$, $\mathcal{S} e$ is the unique span executed by $e$. The function is undefined if the argument event has not completed its span.
- Given spans $a$, $b$, the relation $a \sqsubseteq^\mathcal{S} b$ means that $a$ finished before $b$ started. $\sqsubseteq^\mathcal{S}$ denotes its reflexive closure.
- For a push $u$, $id u$ is the unique id of the node that $u$ inserted into the pool in line 7. Similarly, for a pop $o$, $id o$ is the unique id of the node that $o$ took at the successful CAS in line 38. If $u$ and $o$ have not executed the mentioned lines, $id$ is undefined.
- For a push $u$, $t_s u$ is the abstract timestamp $(id u, t)$, combining $id u$ with the timestamp $t$ that $u$ assigned at line 9. In particular, $t$ is always finite, because newTimestamp only generates finite timestamps. Similarly, for a pop $o$, $t_s o$ is the abstract timestamp $(id o, t)$, combining $id o$ with the timestamp $t$ of the taken node that $o$ read in line 34. Generally, $t_s o$ may return an infinite plain timestamp; however, if elimination pairs are excluded, then timestamps are finite, as explained before. If an event $x$ has not executed its span, $t_s x$ is undefined.
- Abstract timestamps admit the following partial order defined out of $<_T$ on plain timestamps, where we overload the symbol $<_T$ without confusion.

$$(i_1, t_1) <_T (i_2, t_2) \iff t_1 <_T t_2$$

- We define when push $u$ and pop $o$ form an elimination pair.

$$u \text{ ELIM } o \equiv \text{id } u = \text{id } o \wedge u \not\sqsubseteq o$$

In English: (1) $o$ pops the node that $u$ pushed ($id u = id o$), and (2) $u$ and $o$ overlap. Events $u$ and $o$ overlap if $u \not\sqsubseteq o$ and $o \not\sqsubseteq u$, but it is not necessary to explicitly check $o \not\sqsubseteq u$, as that follows from $id u = id o$ and a structural invariant that $o$ cannot pop a node that has not been pushed yet (Appendix B.1 [5]).
- The set of events that occur in elimination pairs is $E \equiv \{ x \mid \exists y. x \text{ ELIM } y \lor y \text{ ELIM } x \}$. As we explicitly exclude elimination pairs from the presentation, we assume that each event variable $x$ occurring in the forthcoming definitions is such that $x \notin E$. In Section 5 we comment how elimination pairs are placed back into consideration.

4.1 Key Abstractions and Invariants

When pop misses a push The key for understanding TS-stacks is explaining what it means for a pop $o$ to have missed a push $u$. Informally, a miss occurs when $o$, in its scan of the pools, takes a push $u'$ with a smaller timestamp than that of $u$ (hence, $u'$ is less recent than $u$). This is critical, because $o$ taking a less recent push than available is seemingly a violation of the LIFO order. However, this does not actually have to be so in the case of TS-stacks, where, for example, it is fine for $u$ to insert into the pool after $o$ has already scanned past the point of insertion. We can say that $u$ occurred too late to really be available for $o$ to pop, and we simply linearize $u$ after $o$. The following definition formalizes when $o$ misses $u$ (i.e., when $u$ occurs too late for $o$), focusing on the atomic timestamp case.

$$\text{Miss } o u \equiv t_s o <_T t_s u \wedge \\
\forall o'. t_s u = t_s o' \implies \text{end}(\mathcal{S} o) < \text{end}(\mathcal{S} o')$$ (1)

The first conjunct directly says that for $o$ to miss $u$, it must be that $o$ takes a node with a smaller timestamp than that of $u$. The second conjunct adds that, intuitively, $u$ remains untaken during the execution of $o$. Indeed, if $u$ is taken by $o'$ ($t_s u = t_s o'$), the definition requires that the span of $o'$ finishes after the span of $o$ ($\text{end}(\mathcal{S} o) < \text{end}(\mathcal{S} o')$). That is, the
CAS that sets the taken flag in the node of $u$ executes after the span of $o$. In other words, if $u$ is taken at all, then it is taken after the span of $o$.

Figure 4 shows a push $u_2$ that overlaps with a pop $o_1$, but $o_1$ takes a push $u_1$ whose timestamp is smaller than that of $u_2$. In our definition, Miss $o_1 u_2$ holds because $u_2$ remains untaken on the stack after $o_1$ terminates. Miss $o_1 u_2$ indicates that we must linearize $o_1$ before $u_2$. And indeed, this is consistent with the situation in the figure, as any order where $u_2$ appears before $o_1$ violates some linearizability requirement. For example, the order $u_1, u_2, o_1$ is sequentially unsound because $o_1$ pops $u_1$ while $u_2$ is the top of the stack, while $u_2, u_1, o_1$ does not respect the ordering of the timestamps of $u_1$ and $u_2$.\(^8\)

Continuing with Figure 4, $o_1$ does not miss $u_3$, even though $u_3$ also overlaps with $o_1$, and $o_1$ takes $u_1$ whose timestamp is smaller than that of $u_3$. In our definition, $\neg$Miss $o_1 u_3$ because the span of $o_3$ ends before the span of $o_1$. $\neg$Miss $o_1 u_3$ indicates no restrictions on the ordering between $o_1$ and $u_3$. For example, the only linearization order of Figure 4 is $u_1, o_1, u_2, u_3, o_3$, but this is forced by the existence of $u_2$. Removing $u_2$, the orders $u_1, u_3, o_1$ (where $u_3$ appears before $o_1$) and $u_1, o_1, u_3, o_3$ (where $u_3$ appears after $o_1$) are both valid.

Misses start late Having defined Miss $o u$, we can now explain the most important invariants of the TS-stack, again focused on the atomic timestamps. The first invariant says that a push $u$ missed by a pop $o$ has a span that starts after the pop's span starts. In other words, a missed push starts after the pop that missed it.

\[ \text{Miss } o u \implies \text{start } (S o) < \text{start } (S u) \quad (2) \]

To intuit why (2) is an invariant, consider a situation when $o$ misses $u$ but $u$’s span starts before $o$’s. In that case, $u$’s pool contains $u$’s node before $o$ even starts its scan. Thus $o$’s scan will encounter $u$ and proceed to either take $u$, or take an even more recent push. At any rate, $o$ will not take a push with a timestamp below that of $u$; thus, $\neg$Miss $o u$.

Disjoint pushes order timestamps The next invariant is that pushes with disjoint spans, produce ordered timestamps. Intuitively, this is so because disjoint push spans make disjoint calls to newTimestamp, which in turn generate ordered timestamps as explained in Section 2.

\[ S u_1 \sqsubseteq^S S u_2 \implies t_s u_1 <_T t_s u_2 \quad (3) \]

\(^8\) We linearize pushes by the order of their timestamps.
4.2 Case: Atomic Timestamps

We next define the visibility \( \preceq \) and separability \( \propto \) relations for atomic timestamps.

\[
u \prec o \equiv t_s u = t_s o \quad (4)
\]

\[
u_1 \propto u_2 \equiv t_s u_1 <_T t_s u_2 \quad (5)
\]

\[
o \propto u \equiv \exists u'. \text{Miss } o u' \land t_s u' \leq_T t_s u \quad (6)
\]

\[
o_2 \propto o_1 \equiv t_s o_1 <_T t_s o_2 \land \neg \exists u'. \text{Miss } o_1 u' \land t_s u' \leq_T t_s o_2 \quad (7)
\]

The definition of \( \prec \) relates \( u \) and \( o \) if they have the same timestamp (i.e., \( o \) took \( u \)). The definition of \( \propto \) comes with three clauses, motivated by the form of the axioms from Figure 3. In particular, we need to separate a push from a push \((u_1 \propto u_2)\), a pop from a push \((o \propto u)\), and a pop from a pop \((o_2 \propto o_1)\), but not a push from a pop, as only the first three clauses of \( \propto \) appear in the axioms.

The clause \( u_1 \propto u_2 \) naturally orders push events according to their timestamps.

The clause \( o \propto u \) extends Miss \( o u' \) to account for pushes being ordered by their timestamps, as per the previous clause. It says that pop \( o \) is separated before push \( u \), if there is a push \( u' \) that was missed by \( o \), and the timestamp of \( u' \) is below (or equals) that of \( u \). For example, in Figure 4 we have \( o_1 \propto u_2 \) and \( o_1 \propto u_3 \).

The clause \( o_2 \propto o_1 \) separates pops inversely to the order of the taken timestamps, or equivalently, inversely to the order of the taken pushes, but under the condition that \( o_1 \) did not miss any push with a timestamp below \( o_2 \). The last requirement is important. For example, if we ignored it in Figure 4, we would obtain \( o_3 \propto o_1 \) since \( t_s u_1 <_T t_s u_3 \). But this order is sequentially unsound; the pushes being ordered as \( u_1, u_2, u_3 \), after \( o_3 \) takes \( u_3 \), the value pushed by \( u_2 \) is at the top of the stack. But then \( o_1 \) cannot execute next, as we need an intervening pop to remove \( u_2 \).

It is worth mentioning that we arrived at the definition of the clause \( o_2 \propto o_1 \) by formal symbol manipulation aimed at fulfilling axiom \( C_1 \) (Concurrent LIFO) after the definitions of the other clauses have been unfolded in \( C_1 \). In hindsight, this may have been expected, as the clauses \( u_1 \propto u_2 \) and \( o \propto u \) are hypotheses of \( C_1 \), while \( o_2 \propto o_1 \) is in the conclusion.

The engineering of the (uniquely determined) definition of the clause \( o_2 \propto o_1 \) thus makes the proof of axiom \( C_1 \) out of definitions (4)-(7) quite straightforward, but for one important observation. Because the axiom contains negations of several clauses of \( \propto \), unfolding the definitions of these clauses reveals comparisons of the form \( t_s x \not<_T t_s y \), where the relation \( <_T \) appears negated. The proof then crucially relies on \( <_T \) being total, so that we can flip the negated comparisons into the form \( t_s y \leq_T t_s x \). It is the requirement of totality of \( <_T \) that makes the described development specific to atomic timestamps. In Section 4.3, we shall see how to adapt to interval timestamps where \( <_T \) is not total.

**Theorem 4.1.** Given \( \prec \) and \( \propto \) as in (4)-(7), the TS-stack with atomic timestamps satisfies the invariants in Section 4.1 and the axioms in Figure 3, and is thus linearizable by Theorem 3.1.

The characteristic part of the proof is showing that the axiom \( C_3 \) (no future dependences) holds, which is where we rely on the invariants (2) and (3). This proof generates obligations, one of which is that \( o \propto u \subset o \) for some push \( u \) and pop \( o \) is impossible, as such \( u \) depends on \( o \) which is in \( u \)'s future. The proof proceeds by contradiction: suppose \( o \propto u \subset o \). By definition of \( o \propto u \), there exists a push \( u' \) missed by \( o \) such that \( t_s u' \leq_T t_s u \). By invariant (2), \( u' \) starts after \( o \) starts, and since \( u \subset o \), it must also be \( u \subset u' \). But then, by invariant (3), it is also \( t_s u <_T t_s u' \). In other words, \( t_s u <_T t_s u' \leq_T t_s u, \) a contradiction.
4.3 Case: Interval Timestamps

The proof from Section 4.2 does not directly apply to the interval timestamps because proving axiom $C_1$ (Concurrent LIFO) relies on the totality of $<_T$ in order to flip the negated inequalities $t_x \not<_T t_y$, which is used to compare the timestamps of two push events in the proofs of the atomic case, at least one of the push events is invariably popped. In other words, the proof does not actually require totality, but only the following weaker property of pop-totality. Formally, if $R$ is a partial order on abstract timestamps, then $R$ is pop-total if:

$$\forall u_1 u_2 u_o (t_s u_1 = t_s u_2) \land (t_s u_1 = t_s u_o) \implies (t_s u_1) R (t_s u_2) \lor (t_s u_2) R (t_s u_1) \lor (t_s u_1 = t_s u_2)$$

In English: if at least one of the pushes is taken, then the timestamps generated by the pushes are totally comparable under $R$.

As an illustration why the weaker property suffices, consider the hypotheses of the axiom $C_1$: these are $u_1 \ll o_1, o_1 \not<_T u_2$ and $u_2 \not<_T u_1$. Let us further assume that $<_T$ in all the definitions is replaced by an arbitrary pop-total $R$. A common pattern throughout the proof of Theorem 4.1 is that three conjuncts of the above form appear together. Such combination entails that $u_1$ and $u_2$ are both popped, thus allowing us to flip any negated relation $R$ in which $t_s u_1$ or $t_s u_2$ may appear.

Indeed, that $u_1$ is popped, and by $o_1$, follows from $u_1 \ll o_1$, which is defined as $t_s o_1 = t_s u_1$. To see that $u_2$ must also be popped consider the following. First, note that $o_1 \not<_T u_2$ implies $\neg$Miss $o_1 u_2$, by an easy derivation. Pushing the negation inside the definition of Miss and substituting $t_s u_1 = t_s o_1$ derives $\neg(t_s u_1) R (t_s u_2) \lor \exists o'. t_s u_2 = t_s o' \land \ldots$. The second disjunct directly says that $u_2$ is popped by some $o'$. By pop-totality of $R$, the first disjunct implies $(t_s u_2) R (t_s u_1) \lor (t_s u_2) = (t_s u_1)$, and thus $t_s u_2 = t_s u_1$, because $\neg(t_s u_2) R (t_s u_1)$ by $u_2 \not<_T u_1$. Thus, $u_1$ and $u_2$ are the same push, and $u_2$ is popped as well.

It follows that we could replicate the atomic case proof to the interval case, if we could replace $<_T$ with some pop-total relation over interval timestamps throughout the definitions and proofs in Sections 4.1 and 4.2. We next define such a relation $\ll$ that includes $<_T$.

$$t_2 \ll t_1 \equiv t_2 <_T t_1 \lor \exists u_1, u_2, t_s u_1 \not<_T t_s u_2 \land \text{TB } u_1 u_2 \land t_2 \leq_T t_s u_2 \land t_s u_1 \leq_T t_1$$

$$\text{TB } u_1 u_2 \equiv \exists o_1. t_s u_1 = t_s o_1 \land \forall o_2. t_s u_2 = t_s o_2 \implies \text{end } (S o_1) < \text{end } (S o_2)$$

The key insight of the definition is that if two pushes $u_1$ and $u_2$ are not already ordered by $<_T$, i.e., $t_s u_1 \not<_T t_s u_2$, we could order their timestamps in $\ll$ in the order in which the pushes are popped. Indeed, if $u_1$ is taken before $u_2$ (TB $u_1 u_2$), then LIFO warrants that $u_2$ is linearized before $u_1$. We thus order $u_2$’s timestamp before $u_1$’s timestamp in $\ll$. It follows that $u_2 \not<_T u_1$ (assuming $\ll$ substitutes $<_T$ in the definition of $u_2 \not<_T u_1$), and consequently that $u_2$ is linearized before $u_1$. The definition of $\ll$ further saturates the relation to include any $t_2 \ll t_1$ where $t_2 \leq_T t_s u_2$ and $t_s u_1 \leq_T t_1$, as then $t_2 \ll t_1$ is forced by $t_s u_2 \ll t_s u_1$.

Returning to taken-before, we define $\text{TB } u_1 u_2$ to hold of two pops $u_1$ and $u_2$ if: (1) $u_1$ is taken and $u_2$ is not, or (2) both are taken by pops $o_1$ and $o_2$, respectively. In the case (2) we require that the span of $o_1$ ends before the span of $o_2$, i.e., $o_1$ took its push before $o_2$ did.

One can now proceed to prove that $\ll$ is a strict partial order that is pop-total, that the invariants “Misses start late” and “Disjoint pushes order timestamps” from Section 4.1,
Visibility and Separability for a Declarative Proof of the Timestamped Stack

continue to hold for the TS-stack with interval timestamps, after substituting \(<_T := \ll\) in the definition of Miss (1), and definitions (2), (3) of the invariants. The visibility and separability relations for the TS-stack with interval timestamps are exactly as in (4)-(7) but with substitution \(<_T := \ll\), and our final theorem about the correctness of TS-stack is obtained simply by retracing the proof of Theorem 4.1.

\textbf{Theorem 4.2.} Let \(<\) and \(\nless\) defined as in (4)-(7) but under the substitution \(<_T := \ll\). The TS-stack with interval timestamps satisfies the invariants in Section 4.1 under the substitution, and the axioms in Figure 3, and is thus linearizable by Theorem 3.1.

5 Discussion, Related and Future Work

Dealing with elimination pairs To handle elimination pairs that were excluded in Section 4, we recursively define indexed families of visibility and separability relations, where \(<_i\) and \(\nless_i\) means that the first \(i\) elimination pairs have been added (Appendix B.1 [5]). At level 0, \(<_0\) and \(\nless_0\) are the relations from Section 4. At some limit level \(n\), where \(n\) is the number of elimination pairs, we have the final relations \(<_n\) and \(\nless_n\) that consider all the events.

The theorems in Section 4 show that the visibility-style axioms in Figure 3 hold for events in \(T \setminus E\), i.e., \(T\) without elimination pairs. They are the base case of our proof in Appendix B.1 [5], which proceeds to inductively show that if the visibility-style axioms hold for the first \(i\) pairs, they continue to hold when the pair \(i + 1\) is added.

Differences with the original algorithm Figure 1 is a simplified version of the algorithm from Appendix B.2 [5]. The latter further treats elimination pair detection and node unlinking (i.e., node deallocation from memory). We consider the simplified version solely for presentation reasons, as the simplification still presents the same verification challenges and suffices to motivate the visibility and separability relations in Section 4. The definitions of these relations transfers to Appendix B.1 [5], where they serve as a basis for defining a family of augmented relations that deal with elimination pairs, as described above.

Having said this, the program that we treat in Appendix B.2 [5] still differs in a relatively minor way from the original program of Dodds et al. [3] in that we elide empty stack detection (i.e. pops returning EMPTY). This can be treated separately as an extra independent step in the proof [7], which means that considering empty pops changes neither the analysis we already presented in Section 4 nor the proof for elimination pairs in Appendix B.1.1.3 [5]. Nevertheless, we plan to augment the proof with an extra step that considers empty pops.

Related proofs Dodds et al. [3] proof is also based on a visibility relation (their \(\text{val}\)), in addition to several other relations. However, our two axiomatizations and proofs differ significantly. Our axiomatization arises from a systematic transformation of a state-based sequential specification of stacks into a history-based concurrent specification, while that of Dodds et al. does not seem to derive from such prior principles, though it does suffice for the linearizability proof. The different axiomatizations give rise to different relations on histories as well. For example, their insert-remove (ir) relation is defined in terms of LPs of submodules. The objective in using LPs of submodules is to start with a definition for that may have linearizability violations, which then gets adjusted along the proof to remove such violations. In contrast, the definitions of our relations in Section 4 require no adjustments since they already lead to a correct linearization, albeit by eliding LPs. As a result, our relations are quite a bit more direct, and support better proof decomposition. In
particular, our proof transfers from the easier atomic timestamp case to the more difficult interval timestamp case, whereas Dodds et al. immediately consider the interval case.

Bouajjani et al. [1] employs forward simulation on the atomic timestamp variant of the TS-stack, but do not attempt the interval timestamp variant. Our proof (Appendix B.2.2 [5]) does not employ simulations, and also lifts the atomic timestamp case to the interval timestamp case. The lifting exploits that the difference between the atomic and interval timestamp cases is not in the program structure, but only in the implementation of $newTimestamp$.

Visibility relations in other contexts Our approach uses visibility and separability relations to model ordering dependencies between events. A general survey of the use of visibility relations in concurrency and distributed systems is given by Viotti and Vukolić [17]. Visibility relations and declarative proofs have also been utilized to specify consistency criteria weaker than linearizability (Emmi and Enea [6]), to introduce a specification framework for weak memory models (Raad et al. [15]), and to specify the RC11 memory model (Lahav et al. [13]).

In contrast to the above papers that focus on the semantics of consistency criteria, our use of visibility relations focuses on verifying specific algorithms and data structures, and is thus closer to the following work where visibility relations are applied to concurrent queues (Henzinger et al. [10, 2]), concurrent stacks (Dodds et al. [3] and Haas [7]), and memory snapshot algorithms (Öhman and Nanevski [14]). We differ from these in the addressed structures, or in the case of Dodds et al. in the structure of the proof and its components.

Our key innovation compared to these works is the introduction of the separability relation and its utilization to systematically axiomatize the stack structure in a novel way.

Visibility and separability as a general methodology The pattern suggested by Sections 3.1 and 3.2, whereby one transforms a history-based sequential specification into a concurrent specification, by replacing the returns-before relation $\prec$ with a separability relation $\searrow$, points towards a general methodology for axiomatizing concurrent structures.

To test the generality of the approach, we have applied it—successfully ([4] and Appendix C.1 [5]—to the RDCSS and MCAS algorithms of Harris et al. [8]. These algorithms write descriptors (a record with information about the task that a thread requires help with) into pointers, so that a thread that reads a descriptor can provide help by executing the described task. These algorithms implicitly “bunch” their help requests into related groups, and the separability relation models gaps between such bunches. On the other hand, the visibility relation models a writer-reader dependency, similarly to the push-pop dependency in this paper. We have also applied the approach to queues, where it derived a mildly streamlined variant of the queue axioms of Henzinger et al. [10, 2], and to locks, including readers-writers locks. In the future, we plan to study if this pattern applies to other concurrent data structures (e.g., memory snapshots, trees, lists, sets, etc.).

References


Visibility and Separability for a Declarative Proof of the Timestamped Stack


