Causal Linearizability: Compositionality for Partially Ordered Executions

Simon Doherty*\(^1\), John Derrick\(^1\), Brijesh Dongol\(^2\), and Heike Wehrheim\(^3\)

\(^1\)Department of Computer Science, University of Sheffield, UK
\(^2\)Department of Computer Science, Brunel University London, UK
\(^3\)Department of Computer Science, University of Paderborn, Germany

February 7, 2018

Abstract

In the interleaving model of concurrency, where events are totally ordered, linearizability is compositional: the composition of two linearizable objects is guaranteed to be linearizable. However, linearizability is not compositional when events are only partially ordered, as in many weak-memory models that describe multicore memory systems. In this paper, we present causal linearizability, a correctness condition for concurrent objects implemented in weak-memory models. We abstract from the details of specific memory models by defining our condition using Lamport’s execution structures. We apply our condition to the C11 memory model, providing a correctness condition for C11 objects. We develop a proof method for verifying objects implemented in C11 and related models. Our method is an adaptation of simulation-based methods, but in contrast to other such methods, it does not require that the implementation totally order its events. We also show that causal linearizability reduces to linearizability in the totally ordered case.

1 Introduction

Linearizability \cite{13, 19} is a well-studied \cite{12} condition that defines correctness of a concurrent object in terms of a sequential specification. It ensures that for each history of an implementation, there is a history of the specification such that (1) each thread makes the same method invocations in the same order, and (2) the order of non-overlapping operation calls is preserved. The condition however, critically depends on the existence of a total order of memory events (e.g., as guaranteed by sequential consistency (SC) \cite{24}) to guarantee contextual refinement \cite{15} and compositionality \cite{13}. Unfortunately most modern systems can only guarantee a partial order of memory events, e.g., due to the effects of relaxed memory \cite{1, 3, 4, 25}. It is known that a naive adaptation of linearizability to the partially ordered setting of weak memory is problematic from the perspective of contextual refinement \cite{13}. In this paper, we propose a compositional generalisation of linearizability for partially ordered executions.

Figs. 1 and 2 show two example\(^*\)\(^1\) of multi-threaded programs on which weak memory model...
effects can be observed. Fig. 1 shows two threads writing to and reading from two shared variables \( x \) and \( y \). Under SC, the \texttt{assert} in process 2 never fails: if \( y \) equals 1, \( x \) must also equal 1. However, in weak memory models like the C11 model [4, 21], this is not true: if the writes to \( x \) and \( y \) are relaxed, process 2 may observe the write to \( y \), yet also observe the initial value \( x \) (missing the write to \( x \) by process 1).

Such effects are not surprising to programmers familiar with memory models [4, 21]. However, programmer expectations for linearizable objects, even in a weak memory model like C11, are different: if the two stacks \( S \) and \( S' \) in Fig. 2 are linearizable, the expectation is that the \texttt{assert} will never fail since linearizable objects are expected to be \textit{compositional} [18, 19], i.e., any combination of linearizable objects must itself be linearizable. However, it is indeed possible for the two stacks to be linearizable (using the classical definition), yet for the program to generate an execution in which the \texttt{assert} fails. The issue here is that linearizability, when naively applied to a weak memory setting, allows too many operations to be considered “overlapping”.

Our key contribution in this paper is the development of a new compositional notion of correctness, called \textit{causal linearizability}, which is defined in terms of an \textit{execution structure} [23], taking two different relations over operations into account: a “precedence order” (describing operations that are ordered in time) and a “communication relation”. Applied to Fig. 2 for a weak memory execution in which the \texttt{assert} fails, the execution restricted to stack \( S \) would not be causally linearizable in the first place. Namely, causal linearizability ensures enough \textit{precedence order} in an execution to ensure that the method call \( S.\texttt{Push}(1) \) occurs before \( S.\texttt{Pop} \), meaning \( S.\texttt{Pop} \) is forced to return 1.

Execution structures are generic, and can be constructed for any weak memory execution that includes method invocation/response events. Our second contribution is one such scheme for mapping executions to execution structures based on the happens-before relation of the C11 memory model. Given method calls \( m_1 \) and \( m_2 \), we say \( m_1 \) precedes \( m_2 \) if the response of \( m_1 \) happens before the invocation \( m_2 \); we say \( m_1 \) communicates with \( m_2 \) if the invocation of \( m_1 \) happens before the response of \( m_2 \).

Our third contribution is a new inductive simulation-style proof technique for verifying causal linearizability of weak memory implementations of concurrent objects, where the induction is over linear extensions of the happens-before relation. This is the first such proof method for weak memory, and one of the first that enables full verification, building on existing techniques for linearizability in SC [26, 12, 8]. Our fourth contribution is the application of this proof technique to causal linearizability of the Treiber Stack in the C11 memory model.

We present our motivating example, the Treiber Stack in C11 in Section 2; describe the problem of compositionality and motivate our execution-structure based solution in Section 3; and formalise causal linearizability and prove compositionality in Section 4. Causal linearizability for C11 is presented in Section 5, and verification of the stack described in Section 6.

## 2 Treiber Stack in C11

The example we consider (see Algorithm 1) is the well-studied Treiber Stack [28], executing in a recent version of the C11 [22] memory model. In C11, commands may be annotated, e.g., \( R \)
Algorithm 1 Release-Acquire Treiber Stack

1: procedure Init
2:   Top := null;
3: procedure Push(v)
4:   n := new(node) ;
5:   n.val := v ;
6: repeat
7:   top := A Top ;
8:   n.nxt := top ;
9: until CAS(&Top, top, n)
10: procedure Pop
11: repeat
12:   repeat
13:     top := A Top ;
14:     until top ≠ null ;
15:     ntop := top.nxt ;
16:     until CAS(&Top, top, ntop) ;
17: return top.val ;

(for release) and A (for acquire), which introduces extra synchronisation, i.e., additional order over memory events [4, 21]. We assume racy read and write accesses that are not part of an annotated command are unordered or relaxed, i.e., we do not consider the effects of non-atomic operations [4]. Full details of the C11 memory model are deferred until Section 5.

Due to weak memory effects, the events under consideration, including method invocation and response events are partially ordered. As we show in Section 3, it turns out that one cannot simply reapply the standard notion of linearizability in this weaker setting; compositionality demands that we use modified form: causal linearizability that additionally requires “communication” across conflicting operations.

In Algorithm 1, all accesses to the shared variable Top are via an annotated command. Thus, any read of Top (lines 7, 13) reading from a write to Top (lines 9, 16) induces happens-before order from the write to the read. This order, it turns out, is enough to guarantee invariants that are in turn strong enough to guarantee causal linearizability of the Stack (see Section 6).

Note that we modify the Treiber Stack so that the Pop operation blocks by spinning instead of returning empty. This is for good reason - it turns out that the standard Treiber Stack (with a non-blocking Pop operation) is not naturally compositional if the only available synchronisation is via release-acquire atomics (see Section 7).

3 Compositionality and execution structures

This section describes the problems with compositionality for linearizability of concurrent objects under weak execution environments (e.g., relaxed memory) and motivates a generic solution using execution structures [23].

Notation. First we give some basic notation. Given a set $X$ and a relation $r \subseteq X \times X$, we say $r$ is a partial order if it is reflexive, antisymmetric and transitive, and a strict order, if it is irreflexive, antisymmetric and transitive. The support of $r$ is denoted $\text{support}(r) = \text{dom}(r) \cup \text{ran}(r)$. A partial or strict order $r$ is a total order iff either $(a, b) \in r$ or $(b, a) \in r$ for all $a, b \in \text{support}(r)$. We typically use notation such as $<, \leq, \prec, \rightarrow$ to denote orders, and write, for example, $a < b$ instead of $(a, b) \in <$.

The operations of an object are defined by a set of labels, $\Sigma$. For concurrent data structures, $\Sigma = \text{Inv} \times \text{Res}$, where $\text{Inv}$ and $\text{Res}$ are sets of invocations and responses (including their input

\footnote{Note that a successful CAS operation comprises both a read and a write access to Top, but we only require release synchronisation here. The corresponding acquire synchronisation is provided via the earlier read in the same operation. This synchronisation is propagated to the CAS by sequenced-before (aka program order), which, in C11, is included in happens-before (see Section 5 for details).}
and return values), respectively. For example, for a stack $S$ of naturals, the \textit{invocations} are given by $\{\text{Push}(n) \mid n \in \mathbb{N}\} \cup \{\text{Pop}\}$, and the \textit{responses} by $\mathbb{N} \cup \{\perp, \text{empty}\}$, and
\[
\Sigma_S = \{(\text{Push}(n), \perp), (\text{Pop}, n) \mid n \in \mathbb{N}\} \cup \{(\text{Pop}, \text{empty})\}
\]
The standard notion of linearizability is defined for a concurrent history, which is a sequence (or total order) of \textit{invocation} and \textit{response} events of operations.

Since operations are concurrent, an invocation of an operation may not be directly followed by its matching response, and hence, a history induces a partial order on operations. For linearizability, we focus on the \textit{real-time} partial order (denoted $\rightarrow$), where, for operations $o$ and $o'$, we say $o \rightarrow o'$ in a history iff the response of operation $o$ \textit{happens before} the invocation of operation $o'$ in the history. A concurrent implementation of an object is linearizable if the real-time partial order ($\rightarrow$) for any history of the object can be extended to a total order that is \textit{legal} for the object’s specification \cite{18}. It turns out that linearizability in this setting is \textit{compositional} \cite{18, 19}: any history of a family of linearizable objects is itself guaranteed to be linearizable.

Unfortunately, histories in modern executions contexts (e.g., due to relaxed memory or distributed computation) are only partially ordered since processes do not share a single global view of time. It might seem that this is unproblematic for linearizability and that the standard definition can be straightforwardly applied to this weaker setting. However, it turns out that a naive application fails to satisfy \textit{compositionality}. To see this, consider the following example.

\textbf{Example 1.} Consider a history $h$, partially ordered by a \textit{happens-before} relation, for two stacks $S$ and $S'$ that are both initially empty (denoted by $\perp$). Suppose that in $h$, the response of $S'.\text{Push}$ happens before the invocation of $S.\text{Pop}$, and the response of $S.\text{Push}$ happens before the invocation of $S'.\text{Pop}$. History $h$ induces a partial order over these operations as shown below:

\[
\begin{align*}
(S'.\text{Pop}, 11) & \rightarrow (S.\text{Push}(42), \perp) \\
(S.\text{Pop}, 42) & \rightarrow (S'.\text{Push}(11), \perp)
\end{align*}
\]

If we restrict the execution above to $S$ only, we can obtain a legal stack behaviour by linearizing $(S.\text{Push}(42), \perp)$ before $(S.\text{Pop}, 42)$ without contradicting the real-time partial order $\rightarrow$ in the diagram above. Similarly, the execution when restricted to $S'$ is linearizable. However, the full execution is not linearizable: ordering both pushes before both pops contradicts the induced real-time partial order ($\rightarrow$ above).

A key contribution of this paper is the development of a correctness condition, \textit{causal linearizability}, that recovers compositionality of concurrent objects with partially ordered histories. Our definition is based on two main insights.

The first insight is that one must augment the real-time partial order with additional information about the underlying concurrent execution. In particular, one must introduce information about the \textit{communication} when linearizing \textit{conflicting} operations. Two operations conflict if they do not commute according to the sequential specification, e.g., for a stack data structure, \textit{Push} and \textit{Pop} are conflicting. Causal linearizability states that for any conflicting operations, say $o$ and $o'$, that are linearized in a particular order, say $o o'$, there must exist some communication from $o$ to $o'$. We represent communication by a relation $\rightarrow$.
Example 2. Consider the partial order in Example 1. For both stacks S and S', the Push must be linearized before the Pop, and hence, we must additionally have communication edges as follows:

\[(S'.Pop(11), S.Push(42)), (S.Pop(42), S'.Push(11))\]

The second insight is that the operations and the induced real-time partial order, \(\rightarrow\), extended with a communication relation, \(\rightarrow\), must form an execution structure \[23\], defined below.

**Definition 3** (Execution structure). Given that \(E\) is a finite set of events, and \(\rightarrow, \rightarrow\subseteq E \times E\) are relations over \(E\), an execution structure is a tuple \((E, \rightarrow, \rightarrow)\) satisfying the following axioms for \(e_1, e_2, e_3 \in E\).

A1 The relation \(\rightarrow\) is a strict order.
A2 Whenever \(e_1 \rightarrow e_2\), then \(e_2 \rightarrow e_1\) and \(\neg(e_2 \rightarrow e_1)\).
A3 If \(e_1 \rightarrow e_2 \rightarrow e_3\) or \(e_1 \rightarrow e_2 \rightarrow e_3\), then \(e_1 \rightarrow e_3\).
A4 If \(e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4\), then \(e_1 \rightarrow e_4\).

Example 4. Consider the execution depicted in Example 2. The requirements of an execution structure, in particular axiom A4 necessitate that we introduce additional real-time partial order edges \(\rightarrow\) as follows.

\[(S.Pop(42), S'.Pop(11)) \rightarrow (S.Push(42), S'.Push(11))\]

For example, the edge \((S.Pop(42), S'.Push(11))\) is induced by the combination of edges \((S.Pop(42), S'.Push(11))\) together with axiom A4.

A consequence of these additional real-time partial order edges is that S (and symmetrically S') is not linearizable since the edge \((S.Pop(42), S'.Push(42))\) must be present even when restricting the structure to S only. Hence compositional no longer fails.

4 Causal linearizability

This section provides a formal definition of causal linearizability, and the compositionality theorem. We define sequential objects in Section 4.1, then define causal linearizability in Section 4.2.

4.1 Sequential specifications

Causal linearizability defines correctness of a concurrent object with respect to a sequential object specification.

**Definition 5** (Sequential object). A sequential object is a pair \((\Sigma, legal)\), where \(legal \subseteq \Sigma^*\) is a prefix-closed sequence of labels.

3The original presentation allows for infinite execution structures, placing a well-foundedness condition on \(\rightarrow\).
For example, in each legal sequence of a stack, each pop operation returns the value from the latest push operation that has not yet been popped, or empty if no such operation exists.

For each sequential object, we define a conflict relation, \( \# \subseteq \Sigma \times \Sigma \), based on the legal behaviours of the object. Two operations conflict if they do not commute in some legal history:

\[
o \# o' = (\forall k_1, k_2 \in \Sigma^*. \ k_1 o o' k_2 \in \text{legal} \iff k_1 o' o k_2 \in \text{legal})
\]

For a stack, we have, for instance, \((\text{Push}(n), \bot)\#(\text{Pop}, n')\) for any \(n, n'\), and for \(n \neq n'\), \((\text{Push}(n), \bot)\#(\text{Push}(n'), \bot)\).

We now show (in Lemma 6 below) that the order of conflicting actions in a sequential history captures all the orders in that history that matter. This is formalized and proved using order relations derived from a legal sequence. However, since the same action can occur more than once in a legal sequence, we lift actions to events by enhancing each action with a unique tag and process identifier. Thus, given a sequential object \(S = (\Sigma, \text{legal})\) an \(S\)-event is a triple \((g, p, a)\) where \(g\) is an event tag (taken from a set of tags \(G\)), \(p\) is a process (taken from a set of processes \(P\)) and \(a\) is a label in \(\Sigma\). We let \(\text{Evt}\) be the set of all \(S\)-events, and for a sequence \(k \in \text{Evt}^*\), \(ev(k)\) be the set of events in \(k\).

The definitions of legality and conflict as well as sequential specifications can naturally be lifted to the level of events by virtue of the action labels. That is, a sequence of events is legal if the sequence of actions it induces is legal. In the following, we therefore use legal to refer to sequences of events and actions of events interchangeably. From a sequence \(k \in \text{Evt}^*\) we derive two relations on events, a temporal ordering \((\rightarrow_k)\) and a causal ordering \((\prec_k)\), where:

\[
e \rightarrow_k e' = \exists k_1, k_2, k_3 \in \text{Evt}^*. k = k_1 e k_2 e' k_3 \quad e \prec_k e' = e \rightarrow_k e' \land ev(k) \neq ev(k')
\]

Any sequential history that extends the causal order of a legal history is itself legal. We formalize this in the following lemma.

**Lemma 6 (Legal linear extensions).** For a sequential object \((\Sigma, \text{legal})\), if \(k \in \text{legal}\) and \(k' \in \text{Evt}^*\), such that \(ev(k) = ev(k')\) and \(\prec_k \subseteq \rightarrow_{k'}\), then \(k' \in \text{legal}\).

**Proof.** We transform \(k\) into \(k'\) by reordering events in \(k\) to match the order \(\rightarrow_{k'}\). We only reorder events that are not conflicting and thus, each step of the transformation preserves legality. This is sufficient to prove that \(k' \in \text{legal}\). Let a mis-ordered pair be any pair of events \(e, e'\) such that \(e \rightarrow_k e'\) but \(e' \rightarrow_{k'} e\). Note that in this case, we have \(e \neq_k e'\), because \(\prec_k \subseteq \rightarrow_{k'}\). Let \(e_1, e_2\) be a mis-ordered pair with minimal distance in \(k\) between the two elements (i.e., so that the number of events in \(k\) between \(e_1\) and \(e_2\) is minimal). We will reorder non-conflicting events in \(k\) so as to eliminate this mis-ordered pair, or reduce its size without creating a new mis-ordered pair. Once all mis-ordered pairs have been eliminated, we will have transformed \(k\) into \(k'\), while preserving the legality of \(k\).

If \(e_1\) and \(e_2\) are adjacent in \(k\), then because \(e_1 \neq_k e_2\), we have \(\neg(e_1 \# e_2)\), and thus we can reorder them to form a new sequence legal with fewer mis-ordered pairs.

If \(e_1 \# e_2\) then we would have \(e_1 \prec_k e_1 \prec_k e_2\) and so \(e_1 \prec_k e_2\), which is a contradiction. The same argument shows that there is no event between \(e_1\) and \(e_2\) that conflicts with both. So let \(k''\) be the sequence derived from \(k\) by reordering \(e_1^1\) forward just past \(e_2\). Note that because there were no conflicts, \(k'' \in \text{legal}\). It remains to show that \(k''\) has no mis-ordered pairs that were not already present in \(k\). This could only happen if there was some \(e_2'\) such that \(e_1^1 \rightarrow_{k'} e_2' \rightarrow_k e_2\) and \(e_1 \rightarrow_{k'} e_2\). Because \(e_1, e_2\) is the mis-ordered pair with minimal gap in \(k\), it must be that \(e_2' \rightarrow_{k'} e_2\), but then \(e_2' \rightarrow_{k'} e_1\) while \(e_1 \rightarrow_k e_2\). Thus, in this case, \(e_1, e_2\) forms a smaller mis-ordered pair, contrary to hypothesis. \(\square\)

---

4Strictly speaking, the process identifier is unimportant for Lemma 6 but we introduce it here to simplify compatibility with the rest of this paper.
As we shall see, this lemma is critical in the proof of our compositionality result, Theorem 10.

4.2 Concurrent executions and causal linearizability

We now define causal linearizability. For simplicity, we assume complete concurrent executions, i.e., executions in which every invoked operation has returned. It is straightforward to extend these notions to cope with incomplete executions.

In general, executions of concurrent processes might invoke operations on several concurrent objects. To capture this, we define a notion of object family, which represents a composition of sequential objects, indexed by some set X.

**Definition 7** (Object family). Suppose X is an index set. For each \( x \in X \), assume a sequential object \( S_x = (\Sigma_x, \text{legal}_x) \) such that \( \Sigma_x \) is disjoint from \( \Sigma_y \) for all \( y \in X \setminus \{x\} \). We define the object family over \( X \), \( S_X = (\Sigma_X, \text{legal}_X) \) by:

- \( \Sigma_X = \bigcup_x \Sigma_x \)
- \( \text{legal}_X = \{ k \in \Sigma_X^* \mid \forall x. k \upharpoonright x \in \text{legal}_x \} \), where \( k \upharpoonright x \) is the sequence \( k \) restricted to actions of object \( x \). Thus the set \( \text{legal}_X \) contains exactly the interleavings of elements of each of the \( \text{legal}_x \).

N.B., the pairwise disjointness requirement on \( \Sigma_x \) can be readily achieved by attaching the object identifier \( x \) to each operation in \( \Sigma_x \).

An execution structure \((E, \rightarrow, \leftarrow)\) is a complete \( S \)-execution structure iff all events in \( E \) are \( S \)-events.

**Definition 8** (Causal linearizability). Let \( S_X = (\Sigma_X, \text{legal}_X) \) be an object family. A complete \( S_X \)-execution structure \((E, \rightarrow, \leftarrow)\) is causally linearizable if there exists a \( k \in \text{legal}_X \) with \( \text{ev}(k) = E \) such that \( \rightarrow \subseteq \leftarrow_k \), and \( \leftarrow \supseteq \leftarrow_k \).

Condition \( \rightarrow \subseteq \leftarrow_k \) ensures that the real-time (partial) order of operations is consistent with the chosen \( k \), while condition \( \leftarrow \supseteq \leftarrow_k \) captures the idea that the causal ordering in \( k \) (i.e., the ordering between conflicting actions) requires a communication in the concurrent execution. Causal linearizability for single objects is a special case of Definition 8 where the family is a singleton set.

To establish compositionality, we must first define an object family’s causal ordering. Note that because an object family’s legal set is just an interleaving of the underlying object’s legal sets, operations from distinct objects can always be reordered, and therefore they never conflict. Thus, we have the following lemma.

**Lemma 9.** Suppose \( S_X = (\Sigma_X, \text{legal}_X) \) is an object family. For any \( k \in \text{legal}_X \), we have \( \leftarrow_k = \bigcup_{x \in X} \leftarrow_k \upharpoonright x \).

For an object family \( S_X = (\Sigma_X, \text{legal}_X) \) and \( x \in X \), we let \( E_x \) be the \( S_X \)-execution structure \( E \) restricted to \( \Sigma_x \).

**Theorem 10** (Compositionality). Suppose \( S_X = (\Sigma_X, \text{legal}_X) \) is an object family over \( X \), and let \( E = (E, \rightarrow, \leftarrow) \) be a complete \( S_X \)-execution structure. Then, \( E_x \) is causally linearizable w.r.t. \( S_x \) for all \( x \in X \) iff \( E \) is causally linearizable w.r.t. \( S_X \).

**Proof.** The implication from right to left is straightforward. For the other direction, for each \( x \in X \), let \( k_x \) be the legal sequential execution witnessing causal linearizability of \( E_x \). Let \( \rightsquigarrow \) be the irreflexive transitive relation defined by

\[
\rightsquigarrow = (\rightarrow \cup \bigcup_{x \in X} \leftarrow_{k_x})^+ \]
We show that $\rightsquigarrow$ is acyclic, and is therefore a strict partial order. Because $\rightsquigarrow$ is a partial order, there is some total order $\rightarrow_m \supseteq \rightsquigarrow$, where $m \in \text{legal}_X$. This total order defines a sequence of labels of $S_X$. We prove that this sequence witnesses the causal linearizability of $E$. By definition, we have $\prec_k z \subseteq \rightsquigarrow \subseteq \rightarrow_m$ for all $x$, and so by Lemma 9 we have $\rightarrow_m | x \in \text{legal}_x$ (where $\rightarrow_m | x$ is the restriction of $\rightarrow_m$ to the events of object $x$). Furthermore,

1. $\rightarrow \subseteq \rightarrow_m$ follows from $\rightsquigarrow \subseteq \rightarrow \subseteq \rightarrow_m$.

2. $\rightarrow \supseteq \prec_k z$ follows from causal linearizability of $E_x$, and hence $\rightarrow \supseteq \bigcup_{x \in X} \prec_k z$, as required.

Thus, $E$ is causally linearizable, as required.

We show that $\rightsquigarrow$ is acyclic by contradiction. Suppose $\rightsquigarrow$ contains a cycle. Pick $P = e_1 \rightsquigarrow e_2 \rightsquigarrow \cdots \rightsquigarrow e_m$ to be the minimal cycle. Since $\rightarrow$ is acyclic, and each $\prec_x$ is acyclic, the cycle $P$ must contain accesses to at least two different objects. Without loss of generality, assume $e_1$ and $e_2$ access different objects, i.e., $e_1 \in E_y$, $e_2 \in E_z$ for some $y \neq z$. Since each $\prec_x$ only orders elements of $E_x$, we must have $e_1 \rightarrow e_2$. Observe that $P$ must be of length greater than two, i.e., it cannot be of the form $e_1 \rightsquigarrow e_2 \rightsquigarrow e_1$ since we would then have $e_1 \rightarrow e_2 \rightarrow e_1$, which contradicts the assumption that $\rightsquigarrow$ is a partial order.

Hence $P$ must contain a third (distinct) element $e_3$. Note that $e_2 \not\rightarrow e_3$, because otherwise we could shorten the cycle $P$, using the transitivity of $\rightarrow$. Thus $e_2, e_3$ are from the same object $z$ and $e_2 \prec_k z$. By the causal linearizability of $E_z$, we must have $e_2 \rightarrow \cdots \rightarrow e_3$. Let $e$ be the element of $P$ following $e_4$ (so possibly $e = e_1$). Note that $e_3 \not\rightarrow_k e$, because otherwise we could shorten the cycle $P$, using the transitivity of $\prec_k z$. Thus, $e_3 \rightarrow e$, so we have

$$e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_3 \rightarrow e$$

By the execution structure axiom A4, we have $e_1 \rightarrow e$, and hence there exists a cycle $e_1 \rightsquigarrow e \rightsquigarrow \cdots \rightsquigarrow e_1$ contradicting minimality of $P$. 

\[\square\]

4.3 Relationship with classical linearizability

In this section, we show that classical linearizability, which is defined for totally ordered histories of invocations (events of type $\text{Inv}$) and responses (events of type $\text{Res}$), degenerates to causal linearizability. As in the previous section, for simplicity, we assume the histories under consideration are complete; extensions to cope with incomplete histories are straightforward.

First, we describe a method, inspired by the execution structure constructions given by Lamport [23], for constructing execution structures for any well-formed partially ordered history. We let $\text{Hist} = (\text{Inv} \cup \text{Res}) \times (\text{Inv} \cup \text{Res})$ denote the type of all histories. A history is well-formed if it is a partial order and the history restricted to each process is a total order of invocations followed by their matching response. The set of all matching pairs of invocations and responses in a history $h$ is given by $\text{mp}(h)$. A history is sequential iff it is totally ordered and each invocation is immediately followed by its matching response. Note that a history could be totally ordered, but not sequential (as is the case for the concurrent histories considered under SC [13, 13]).

Definition 11. Let $h \subseteq \text{Hist}$ be a well-formed (partially ordered) history. We say $\text{exec}(h) = (E, \rightarrow, \rightsquigarrow)$ is the execution structure corresponding to $h$ if

\[
E = \text{mp}(h)
\]

\[
\rightarrow = \{(i_1, r_1), (i_2, r_2) \in E \times E \mid (r_1, i_2) \in h\}
\]

\[
\rightsquigarrow = \{(i_1, r_1), (i_2, r_2) \in E \times E \mid (i_1, r_2) \in h\}
\]
We now work towards the standard definition of linearizability. Recall that a sequential object (see Definition 3) is defined in terms of sequences of labels of type $\Sigma^*$, where $\Sigma = Inv \times Res$, whereas sequential histories are of type $Hist$. Thus, we define a function $\gamma : Hist \rightarrow \Sigma^*$ such that for each pair $(i, r), (i', r') \in mp(hs)$ of sequential history $hs$ we have $(r, r') \in hs$ iff $(i, r) \rightarrow_{\gamma(hs)} (i', r')$. Thus, the order of operations in $hs$ and $\gamma(hs)$ are identical.

A complete history $h$ is linearizable w.r.t. a (family of) sequential object(s) $S = (\Sigma, legal)$ iff there exists a sequential history $hs$ such that $\gamma(hs) \in legal$, for each process $p$, $h \upharpoonright p = hs \upharpoonright p$ and $h \subseteq hs$ [13].

**Theorem 12.** Suppose $h$ is a totally ordered complete history and $S$ a (family of) sequential object(s). Then $h$ is linearizable w.r.t. $S$ iff exec($h$) is causally linearizable w.r.t. $S$. 

5 Causal linearizability of C11 implementations

We now introduce the C11 memory model, where we adapt the programming-language oriented presentation of C11 [21, 10], but we ignore various features of C11 not needed for our discussion, including non-atomic operations and fences.

**The C11 memory-model.** Let $L$ be a set of locations (ranged over by $x, y$), let $V$ be a set of values (ranged over by $u, v$). Our model employs a set of memory events, which can be partitioned into read events, $R$, write events, $W$, and update events, $U$. Moreover, let $Mod = W \cup U$ be the set of events that modify a location, and $Qry = R \cup U$ be the set of events that query a location. For any memory event $e$, let $loc(e)$ be the event’s location, and let $ann(e)$ be the event’s annotation. Let $Loc(x) = \{ e \mid loc(e) = x \}$. For any query event let $rval(e)$ be the value read; and for any modification event let $wval(e)$ be the value written. An event may carry a synchronisation annotation, which may either be a release, $R$, or an acquire, $A$, annotation.

A C11 execution (not to be confused with an execution structure) is a tuple $D = (D, sb, rf, mo)$ where $D$ is a set of events, and $sb, rf, mo \subseteq D \times D$ define the sequence-before, reads-from and modification order relations, respectively. We say a C11 execution is valid when it satisfies:

(V1) $sb$ is a strict order, such that, for each process $p$, the projection of $sb$ onto $p$ is a total order;

(V2) for all $(w, r) \in rf$, $loc(w) = loc(r)$ and $wval(w) = rval(r)$;

(V3) for all $r \in D \cap Qry$, there exists some $w \in D \cap Mod$ such that $(w, r) \in rf$;

(V4) for all $(w, w') \in mo$, $loc(w) = loc(w')$; and

(V5) for all $w, w' \in W$ such that $loc(w) = loc(w')$, $(w, w') \in mo$ or $(w', w) \in mo$.

Other relations can be derived from these basic relations. For example, assuming $D_R$ and $D_A$ denote the sets of events with release and acquire annotations, respectively, the synchronises-with relation, $sw = rf \cap (D_R \times D_A)$, creates interthread ordering guarantees based on synchronisation annotations. The from-read relation, $fr = (rf^{-1}; mo) \setminus Id$, relates each query to the events in modification order after the modification that it read from. Our final derived relation is the happens before relation $hb = (sb \cup sw)^+$, which formalises causality. We say that a C11 execution is consistent if

(C1) $hb$ is acyclic, and

(C2) $hb; (mo \cup rf \cup fr)$ is irreflexive.
Method invocations and responses. So far, the events appearing in our model are standard. Our goal is to model algorithms such as the Treiber stack. Thus, we add method events to the standard model, namely, invocations, Inv, and responses, Res. Unlike weak memory at the processor architecture level, where program order may not be preserved\[13\], program order in C11 is consistent with happens-before order, and hence, these can be introduced here in a straightforward manner. The only additional requirement is that validity also requires (V6) sb for each process projected restricted Inv \cup Res must be an alternating sequence of invocations and matching responses, starting with an invocation.

Dynamic memory. To describe the behaviour of algorithms, such as the Treiber Stack, we must define reads and writes to higher-level structures. To this end, we develop a simple theory of references to objects, the fields of those objects and memory allocations for the object. We let F be the set of all fields and A be the set of all memory allocation events, which is an event of the form A(l) for a location l. We let . : L \times F \rightarrow L be the function that returns a location for a given location, field pair. We use infix notation x.f for .(x, f), where x \in L and f \in F. We then introduce three additional constraints: (A1) for every a, a' \in E \cap A, if loc(a) = loc(a') then a = a'; and (A2) if l.f = l'.f' then l = l' and f = f'. (A3) for all locations l and fields f there are no allocations of the form A(l.f).

From C11 executions to execution structures. A C11 execution with method invocations and responses naturally gives rise to an execution structure. First, for a C11 execution F, let the history of D, denoted hist(D) be the happens-before relation for D restricted to the invocation and response events. By (V6), hist(D) is a well-formed history. Thus, we can apply the construction defined in Section 4.31 to build an execution structure exec(hist(D)).

Definition 13. We say that a C11 execution D is causally linearizable w.r.t a sequential object if exec(hist(D)) is.

We can now state a compositionality result for a C11 execution D of an object family X. The property follows from Theorem 10 and the fact that for any object x \in X, exec(hist(D_x)) = exec(hist(D))_x, where D_x is D restricted to events of object x. Note that D_x contains all events of x, i.e., all invocations, responses and low-level memory operations of x.

Corollary 14 (Compositionality for C11 executions). Suppose that S_X = (\Sigma_X, legal_X) is an object family over X, and let D be an execution. Then, D_x is causally atomic w.r.t. S_x for all x \in X iff \exists D \text{ is causally atomic w.r.t. } S_X.

Finally, note that because the sb relation is included in hb, hist(D) includes program order on the invocations and responses of D.

6 Verification

We now describe an operational method for proving that a given C11 execution is causally linearizable w.r.t a given sequential object. Accordingly, we give a state-based, operational model of a sequential object that generates legal sequences of labels (Definition 5), then present a simulation-based proof rule for causal linearizability (Section 6.1). Then, we illustrate our technique on the Treiber Stack (Section 6.2).

6.1 A simulation relation over happens-before

An operational sequential object is a tuple (\Gamma, init, \tau) where: \Gamma is a set of states; init \in \Gamma is the initial state and \tau : \Gamma \times H \times Inv \rightarrow \Gamma \times H, where H = (Inv \times Res)^* is a partial update function.
that applies an invocation to a state and a history, returning the resulting state and updated history. We require that for \( s \in \Gamma \), \( h \in H \) and \( i \in \text{Inv} \), there exists some \( r \in \text{Res} \), such that \( \tau(s, h, i) = (s, h \cdot \langle (i, r) \rangle) \), where we use \( \cdot \) for sequence concatenation. This response \( r \) is the object’s response to the invocation \( i \).

**Example 15** (Operational sequential stack). A stack containing natural numbers can be represented as an operational sequential object in the following way. Let \( \Gamma = \mathbb{N}^* \), \( \text{init} = \langle \rangle \) and define the update function as follows

\[
\tau(s, h, \text{Push}(n)) = (n \cdot s, h \cdot \langle \text{Push}(n), \bot \rangle) \quad \tau(n \cdot s, h, \text{Pop}) = (s, h \cdot \langle \text{Pop}, n \rangle)
\]

for \( n \in \mathbb{N} \) and \( s \in \Gamma \). Note that assuming \( i \) is a stack invocation (as per Section 3), \( \tau(s, i) \) is defined iff \( s \neq \langle \rangle \) or \( i \neq \text{Pop} \).

Given an operational sequential object \( \Sigma = (\Gamma, \text{init}, \tau) \), it is easy to construct a corresponding sequential object (in the sense of Definition 5). Let \( \Sigma_\Sigma = \text{Inv} \times \text{Res} \) and let \( \text{legal}_\Sigma \) be the set of histories returned by \( \tau \). Thus \( (\Sigma_\Sigma, \text{legal}_\Sigma) \) is a sequential object, and our method verifies causal linearizability w.r.t that object.

For the remainder of this section, fix a C11 execution \( \mathcal{D} = (D, s_b, rf, mo) \) and an operational sequential object \( \mathcal{S} = (\Gamma, \text{init}, \tau) \). We describe a method for proving that \( \mathcal{D} \) is causally linearizable w.r.t \( \mathcal{S} \). Our proof method is an induction on the length of some linear extension of \( \mathcal{D} \)'s happens-before order. The proof proceeds by remembering the set \( Z \subseteq D \) of events that have already been considered by the induction, i.e., \( Z \) defines the current stage of the induction. The set \( Z \) is assumed to be downclosed with respect to \( hb \), i.e., if \( z \in Z \) and \( (z', z) \in hb \), then \( z' \in Z \). At each stage of the induction, we add an arbitrary \( e \notin Z \) to \( Z \), where \( e \)'s \( hb \) predecessors are already in \( Z \) (i.e., the set \( Z \cup \{e\} \) is also downclosed w.r.t. \( hb \)).

Correctness of each inductive step is formalised by a simulation relation, \( \rho \), relating the events in the current state, \( Z \), to a state of the operational sequential object. Each inductive step of the implementation must match a “move” of the sequential object, i.e., be a stutter step, or a state update as given by the update function of the sequential object. Moreover, assuming that \( \rho \) holds for \( Z \) (before each inductive step), \( \rho \) must hold after the step (i.e., for \( Z \cup \{e\} \)).

Following the existing verification literature [12], we refer to events corresponding to non-stuttering steps as linearisation points: the points where the high-level operation appears to take effect. The verifier must define a function \( lp : D \cap \text{Inv} \rightarrow D \) to determine the memory event that linearizes the given invocation, and this function must satisfy certain constraints with respect to the simulation relation \( \rho \), as described in Definition 14 below.

For each low-level operation, we must also determine the invocation and response to which it belongs. Thus we also define a function \( \mu : D \rightarrow D \cap \text{Inv} \) that maps each event in \( D \) to the invocation responsible for producing \( e \), and a function and \( \nu : D \rightarrow D \cap \text{Res} \) that maps \( e \) to the response produced by \( e \)'s invocation. More formally, \( \mu(e) \) is the latest invocation in \( sb \)-order prior to \( e \), and \( \nu(e) \) is the earliest response in \( sb \)-order after \( e \).

Thus, we obtain the following definition.

**Definition 16** (hb-simulation). Suppose \( \mathcal{D} = (D, s_b, rf, mo) \) is an execution and \( \mathcal{S} = (\Gamma, \text{init}, \tau) \) an operational sequential object. An hb-simulation is a relation \( \rho \subseteq 2^D \times (\Gamma \times H) \) such that:

1. \( \rho(\emptyset, (\text{init}, \langle \rangle)) \), and

(\text{initialisation})

2. for all \( Z \subseteq D \), and events \( e \in D \setminus Z \) such that \( Z \cup \{e\} \) is down-closed w.r.t \( \mathcal{D} \)'s happens-before order, if \( \rho(Z, (s, h)) \) then

(\text{stutter step})

\[
\begin{align*}
&\text{if } e \neq lp(\mu(e)) \text{ then } \\
&\rho(Z \cup \{e\}, (s, h))
\end{align*}
\]
(b) if $e = \text{lp}(\mu(e))$ (linearization step)

provided $i = \mu(e)$, $h' = h \cdot ((i, r))$, and $\tau(s, h, i) = (s', h')$, then

i. $\rho(Z \cup \{e\}, (s', h'))$, and

ii. $\nu(e) = r$, and

iii. for all operations $(i', r')$ in $h$, if $(i', r') \prec_{h'} (i, r)$ then $(\text{lp}(i'), e) \in \text{hb}$.

The initialisation is straightforward, while the two inductive steps consider a new $e$ for inclusion in $Z$ following hb order. If $e$ is a stutter step, we only have to prove that $\rho$ is preserved by adding $e$ to $Z$. If $e$ is a linearization step (that is, if $e = \text{lp}(\mu(e))$), then there are three obligations: prove that $\rho$ is preserved \[2\{b\}ii\]; prove that the response of the high-level operation matches that returned by the sequential object \[2\{b\}iii\]; and prove that whenever some operation that has already been linearized is causally prior to the newly linearized operation, then that operation’s linearization point is hb-prior to the new event $e$ \[2\{b\}iii\].

**Theorem 17** (Soundness of hb-simulation). If $\rho$ is an hb-simulation for a C11 execution $\mathbb{D}$, then $\mathbb{D}$ is causally linearizable.

**Proof.** The proof below uses a formulation of an operational sequential object where $\tau$ that does not maintain a history.

Fix the operational sequential object $S = (\Gamma, \text{init}, \tau)$. Fix the execution $\mathbb{D}$, and let $\leq_E$ be any linear extension of $\mathbb{D}$’s hb relation. Assume that $\text{lp}$ is the linearization function and $\rho$ is the simulation relation.

We perform an induction on the indexes of $\leq_E$. Let $e_n$ be the nth event in $\leq_E$ order, so we are indexing from 0. Let $Z_n$ be the set of events strictly below the nth index. Thus,

$$Z_n = \{e_m | m < n\}$$

Note that $Z_0 = \emptyset$ and $Z_{n+1} = Z_n \cup \{e_n\}$. We define a function $\text{rep} : \{n | n < | \leq_E |\} \rightarrow \Gamma$ recursively as follows:

$$\text{rep}(0) = \text{init}$$

$$\text{rep}(n + 1) = \text{rep}(n) \quad \text{when } \text{lp}(\mu(e_n)) \neq e_n \tag{2}$$

$$\text{rep}(n + 1) = \pi_1(\tau(\text{rep}(n), \mu(e_n))) \quad \text{when } \text{lp}(\mu(e_n)) = e_n \tag{3}$$

By induction, we have $\rho(Z_n, \text{rep}(n))$ for all $n < | \leq_E |$.

- Because $\rho(Z_0, \text{rep}(0)) = \rho(\emptyset, \text{init})$, Proposition [1] ensures that $\rho(Z_0, \text{rep}(0))$.

- Assume $\rho(Z_n, \text{rep}(n))$ and $\text{lp}(\mu(e_n)) \neq e_n$. Then $\rho(Z_{n+1}, \text{rep}(n+1)) = \rho(Z_n \cup \{e_n\}, \text{rep}(n))$, and thus Property [2] ensures that $\rho(Z_{n+1}, \text{rep}(n + 1))$.

- Assume $\rho(Z_n, \text{rep}(n))$ and $\text{lp}(\mu(e_n)) = e_n$.

Then $\rho(Z_{n+1}, \text{rep}(n + 1)) = \rho(Z_n \cup \{e_n\}, \pi_1(\tau(\text{rep}(n), \mu(e_n))))$, and thus Property [2] ensures that $\rho(Z_{n+1}, \text{rep}(n + 1))$.

We turn now to defining $k$, the legal sequence we need to witness causal linearizability of $\mathbb{D}$.

$$k_0 = \langle \rangle \tag{4}$$

$$k_{n+1} = k_n \quad \text{when } \text{lp}(\mu(e_n)) \neq e_n \tag{5}$$

$$\text{rep}(n + 1) = k_n \cdot ((\mu(e), \nu(e))) \quad \text{when } \text{lp}(\mu(e_n)) = e_n \tag{6}$$
It is easy to see that this is a legal history, and that \((k_n, rep(n))\) is a move.

We need to show that \(\leq_H \subseteq \rightarrow_k\). Consider a response \(r\) and invocation \(i\) such that \((r, i) \in hb\).

Let \(i'\) be the invocation of \(r\), and let \(r'\) be the response of \(i\). Because \(\mu(lp(i)) = i\), we have \(\{(lp(i'), r), (i, lp(i))\} \subseteq sb \subseteq hb\), and thus \((lp(i'), lp(i)) \in hb\) and so \(lp(i')\) appears at an earlier point in \(\leq_E\) than \(lp(i)\), and therefore \((i', r) \rightarrow_k (i, r')\), as required.

Finally, we must show that \(\varsigma_k \subseteq \rightarrow_{\mathcal{V}_D}\). This is a simple induction on the length of \(k\), with the hypothesis that, for all operations \((i, r), (i', r') \in k\), \((i, r) \preceq_k (i', r')\) then \((lp(i), lp(i') \in hb\).

At each step we apply Property 2(b)iii. Thus, for each existing operation \((i, r)\) and new operation \((i', r')\), we have \((i, r) \preceq_k (i', r')\) \(\Rightarrow (lp(i), lp(i') \in hb\) immediately. On the other hand, \((i', r') \not\preceq_k (i, r)\) is impossible, because \((i', r') \rightarrow_k (i, r)\) is false.

This completes our proof. \(\square\)

### 6.2 Case-study: the Treiber Stack

We now outline an \(hb\)-simulation relation \(\rho\) for the Treiber stack. We fix some arbitrary C11 execution \(D = (D, sb, rf, mo)\) that contains an instance of the Treiber stack. That is, the invocations in \(D\) are the stack invocations, and the responses are the stack responses (as given in Section 6). Furthermore, the low-level memory operations between these invocations and responses are generated by executions of the operations of the Treiber stack (Algorithm 1).

The main component of our simulation relation guarantees correctness of the data representation, i.e., the sequence of values formed by following next pointers starting with \&Top forms an appropriate stack, and we focus on this aspect of the relation. As is typical with verifications of shared-memory algorithms, there are various other properties that would need to be considered in a full proof.

In a sequentially consistent setting, the data representation can easily be obtained from the state (which maps locations to values). However, for C11 executions calculating the data representation requires a bit more work. In what follows, we define various functions that depend on a set \(Z\) of events, representing the current stage of the induction.

We define the latest write in \(Z\) to a location \(x\) as \(\text{latest}_Z(x) = \max(mo)(Z \cap \text{Loc}(x))\) and the current value of a location \(x\) in some set \(Z\) as \(\text{cval}_Z(x) = \text{wval}(\text{latest}_Z(x))\), which is the value written by the last write to \(x\) in modification order. It is now straightforward to construct the sequence of values corresponding to a location as \(\text{stackOf}_Z(x) = v \cdot \text{stackOf}_Z(y)\), where \(v = \text{cval}_Z(x.val)\) and \(y = \text{cval}_Z(x.nxt)\).

Now, assuming that \((s, h)\) is a state of the operational sequential stack, our simulation requires:

\[
\text{stackOf}_Z(\text{cval}_Z(\&Top)) = s
\]  

Further, we require that all modifications of \&Top are totally ordered by \(hb\):

\[
\forall m, m' \in Z \cap \text{Mod}(\&Top). \ (m, m') \in hb \lor (m', m) \in hb
\]

(8) to ensure that any new read considered by the induction sees the most recent version of \&Top.

The linearization function \(lp\) for the Treiber stack is completely standard: each operation is linearized at the unique update operation generated by the unique successful CAS at line 9 (for pushes) or line 16 (for pops).

In what follows, we illustrate how to verify the proof obligations given in Definition 16 for the case where the new event \(e\) is a linearization point. Let \(e\) be an update operation that is generated by the CAS at line 9 of the push operation in Algorithm 1. The first step is to prove that every modification of \&Top in \(Z\) is happens-before the update event \(e\). Formally,

\[
\forall m \in Z \cap \text{Mod} \cap \text{Loc}(\&Top). \ (m, e) \in hb
\]  

(9)

13
Proving this formally is somewhat involved, but the essential reason is as follows. Note that there is an acquiring read \( r \) to \&\( \text{Top} \) executed at line 7 of \( e \)'s operation and \( \text{sb}-\)prior to \( e \). \( r \) reads from some releasing update \( u \). Thus, by Property 8, and the fact the \( \text{hb} \) contains \( \text{sb} \), \( e \) is happens after \( u \), and all prior updates. If there were some update \( u' \) of \&\( \text{Top} \) such that \( (u', e) \notin \text{hb} \), then \( (u', u) \notin \text{hb} \) so by Property 8, \( (u, u') \in \text{hb} \). But it can be shown in this case that the CAS that generated \( e \) could not have succeeded, because \( u' \) constitutes an update intervening between \( r \) and \( e \). Therefore, there can be no such \( u' \).

Property 9 makes it straightforward to verify that Condition 2(b) iii of Definition 16 is satisfied. To see this, note that every linearization point of every operation is a modification of \&\( \text{Top} \). Thus, if \((i', r')\) is some operation such that \( \text{lp}(i') \in Z \) (so that this operation has already been linearized) then \((\text{lp}(i'), e) \in \text{hb} \).

Using Property 9 it is easy to see that both Property 6 and Property 8 are preserved. We show by contradiction that \( \text{latest}_{Z'}(\&\text{Top}) = e \). Otherwise, we have \((e, \text{latest}_{Z'}(\&\text{Top})) \in \text{mo} \). Therefore \((\text{latest}_{Z'}(\&\text{Top}), e) \notin \text{hb} \), but \( \text{latest}_{Z'}(\&\text{Top}) \) is a modification operation, so this contradicts Property 9.

It follows from \( \text{latest}_{Z'}(\&\text{Top}) = e \) that \( \text{stackOf}(\text{cval}_{Z'}(\&\text{Top})) = \text{stackOf}(\text{wval}(e)) \). Given this, it is straightforward to show that Property 6 is preserved. This step of the proof relies on certain simple properties of push operations. Specifically, we need to show that the current value of the \text{val} field of the node being added to the stack (formally, \( \text{cval}_{Z'}((\text{wval}(e)).\text{nxt}) \)) is the value passed to the push operation; and that the current value of the \text{nxt} field (formally, \( \text{cval}_{Z'}((\text{wval}(e)).\text{nxt}) \)) is the current value of \&\( \text{Top} \) when the successful CAS occurs. These properties can be proved using the model of dynamic memory given in Section 5.

7 A synchronisation pitfall

We now describe an important observation regarding failure of compositionality of read-only operations caused by weak memory effects. The issue can be explained using our abstract notion of an execution structure, however, a solution to the problem is not naturally available in C11 with only release-acquire annotations.

Consider the Treiber Stack in Algorithm 1 that returns empty instead of spinning; namely where the inner loop (lines 12-14) is replaced by code block “top := A Top ; if top = null then return empty”. Such an implementation could produce executions such as the one in Fig. 3 which, like the examples in Section 3, is not compositional. Recovering compositionality requires one to introduce additional communication edges as shown in Fig. 4. In the C11 memory model, these correspond to “from-read” anti-dependencies from a read to a write overwriting the value read. However, release-acquire synchronisation is not adequate for promoting from-read order in the memory to happens-before.

One fix would be to disallow read-only operations, e.g., by introducing a release-acquire CAS operation on a special variable that always succeeds at the start of each operation. However, such a fix is somewhat unnatural. Another would be to use C11’s \text{SC} annotations, which can induce
synchronisation across from-read edges. However, the precise meaning of these annotations is still a topic of active research [22] [6].

8 Conclusion and related work

We have presented causal linearizability, a new correctness condition for objects implemented in weak-memory models, that generalises linearizability and addresses the important problem of compositionality. Our condition is not tied to a particular memory model, but can be readily applied to memory models, such as C11, that feature a happens-before relation. We have presented a proof method for verifying causal linearizability. We emphasise that our proof method can be applied directly to a standard axiomatic memory model. Unlike other recent proposals [11] [20], we model C11’s relaxed accesses without needing to prohibit their problematic dependency cycles (so called “load-buffering” cycles).

Although causal linearizability has been presented as a condition for concurrent objects, we believe it is straightforward to extend this condition to cover, for example, transactional memory. We intend to develop our approach into a framework in which the behaviour of programs that mix transactional memory, concurrent objects and primitive weak-memory operations can be precisely described in a compositional fashion.

Causal linearizability is closely related to causal hb-linearizability defined in [13], which is a causal relaxation of linearizability that uses specifications strengthened with a happens-before relation. The compositionality result there requires that either a specification is commuting or that a client is unobstructive (does not introduce too much synchronisation). Our result is more general as we place no such restriction on the object or the client. Others [9] define a correctness condition, also called causal linearizability, that is only compositional when the client satisfies certain constraints; in contrast, we achieve full decoupling. Furthermore, that condition is only defined when the underlying memory model is given operationally, rather than axiomatically like C11. Early attempts, targetting TSO architectures, used totally ordered histories but allowed the response of an operation to be moved to a corresponding “flush” event [16] [7] [27] [14]. Others have considered the effects of linearizability in the context of a client abstraction. This includes a correctness condition for C11 that is strictly stronger than linearizability under SC [5]. Although we have applied causal linearizability to C11, causal linearizability itself is more general as it can be applied to any weak memory model with a happens-before relation. Causal consistency [2] is a related condition, aimed at shared-memory and data-stores, which has no notion of real-time order and is not compositional.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. IEEE Computer, 29(12):66–76, 1996.

[2] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. Distributed Computing, 9(1):37–49, Mar 1995.

[3] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst., 36(2):7:1–7:74, 2014.

[4] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In Thomas Ball and Mooly Sagiv, editors, POPL, pages 55–66. ACM, 2011.
[5] Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. In Roberto Giacobazzi and Radhia Cousot, editors, POPL, pages 235–248. ACM, 2013.

[6] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and opencl. In POPL, pages 634–648. ACM, 2016.

[7] Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Concurrent library correctness on the TSO memory model. In Helmut Seidl, editor, ESOP, volume 7211 of Lecture Notes in Computer Science, pages 87–107. Springer, 2012.

[8] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In FORTE, volume 3235 of Lecture Notes in Computer Science, pages 97–114. Springer, 2004.

[9] Simon Doherty and John Derrick. Linearizability and causality. In SEFM, volume 9763 of Lecture Notes in Computer Science, pages 45–60. Springer, 2016.

[10] Marko Doko and Viktor Vafeiadis. A program logic for C11 memory fences. In VMCAI, volume 9583 of Lecture Notes in Computer Science, pages 413–430. Springer, 2016.

[11] Marko Doko and Viktor Vafeiadis. Tackling real-life relaxed concurrency with FSL++. In ESOP, pages 448–475, 2017.

[12] B. Dongol and J. Derrick. Verifying linearisability: A comparative survey. ACM Comput. Surv., 48(2):19:1–19:43, 2015.

[13] B. Dongol, R. Jagadeesan, J. Riely, and A. Armstrong. On abstraction and compositionality for weak-memory linearisability. In VMCAI, volume 10747 of Lecture Notes in Computer Science, pages 183–204. Springer, 2018.

[14] Brijesh Dongol, John Derrick, and Graeme Smith. Reasoning algebraically about refinement on TSO architectures. In ICTAC, volume 8687 of Lecture Notes in Computer Science, pages 151–168. Springer, 2014.

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

[16] Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Show no weakness: Sequentially consistent specifications of TSO libraries. In Marcos K. Aguilera, editor, DISC, volume 7611 of Lecture Notes in Computer Science, pages 31–45. Springer, 2012.

[17] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In Siddhartha Chatterjee and Michael L. Scott, editors, PPoPP, pages 175–184. ACM, 2008.

[18] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. ACM TOPLAS, 12(3):463–492, 1990.

[19] Maurice Herlihy and Nir Shavit. The art of multiprocessor programming. Morgan Kaufmann, 2008.

[20] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in iris. In ECOOP, pages 17:1–17:29, 2017.
[21] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In Rastislav Bodik and Rupak Majumdar, editors, *POPL*, pages 649–662. ACM, 2016.

[22] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *PLDI*, pages 618–632. ACM, 2017.

[23] L. Lamport. On interprocess communication. part I: basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[24] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[25] J. Manson, W.Pugh, and S. V. Adve. The Java memory model. In *POPL*, pages 378–391. ACM, 2005.

[26] Gerhard Schellhorn, John Derrick, and Heike Wehrheim. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. Comput. Log.*, 15(4):31:1–31:37, 2014.

[27] Oleg Travkin and Heike Wehrheim. Handling TSO in mechanized linearizability proofs. In *Haifa Verification Conference*, volume 8855 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2014.

[28] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Res. Ctr., 1986.
A Potentially incomplete executions

A complication with concurrent executions is that they may contain incomplete operations (that have been invoked, but have not yet returned). Since the effect of an incomplete operation may be globally visible, they cannot simply be ignored. This phenomenon has been well studied and arises in the definitions of linearizability [18] and opacity [17]. This section describes how we cope with incomplete operations in the context of causal linearizability.

We define the completable extension of a sequential object \( S = (\Sigma, \text{legal}) \) to be a triple \( T = (S, I, C) \), where \( I \) is a set of allowable incomplete actions and \( C : I \to 2^\Sigma \) is a completion function that maps each \( I \) to a set of possible completions for \( I \).

Example 18. If \( S = (\Sigma, \text{legal}) \) is a concurrent object the set of allowable incomplete operations and completion function is defined by:
\[
I = \{ i \in \text{Inv} \mid \exists k_1, k_2 \in \Sigma^* \text{ s.t. } k_1(i, r)k_2 \in \text{legal} \} \\
C = \lambda i. \{ (i, r) \in \Sigma \}
\]

For the Treiber stack in Algorithm 1, we have \( I_S = \{ \text{Push}(n) \mid n \in \mathbb{N}\} \cup \{ \text{Pop} \} \), and \( C_S(\text{Push}(n)) = \{ \bot \} \) and \( C_S(\text{Pop}) = \mathbb{N} \cup \{ \text{empty} \} \).

A completable extension of an object family \( S_X \) is a triple \( T_X = (S_X, I_X, C_X) \), where \( I_X = \bigcup_x I_x \) and \( C_X(a) = C_x(a) \), with \( x \) being the unique element of \( X \) such that \( a \in \Sigma_x \).

We say that \( E = (E, \rightarrow, \rightarrow') \) is a \( T \)-execution structure iff \( E \subseteq \Sigma \cup I \) such that \( \text{dom}(\rightarrow) \cap I = \emptyset \), i.e., no element of \( E \) may depend (in real-time order) on an element in \( I \). Note that there may be \( \rightarrow \) edges both in and out of elements in \( E \cap I \) and \( \rightarrow \) edges into \( E \cap I \). A \( T \)-execution structure is causally atomic if we can replace all incomplete events by complete events in a way that is allowed by the corresponding sequential object. This process is analogous to the extension of incomplete histories to complete histories, as allowed by linearizability and opacity in the classical (i.e., sequentially consistent) setting.

Definition 19 (Causal linearizability). Let \( S = (\Sigma, \text{legal}) \) be a family of sequential objects and \( T \) its completable extension. A \( T \)-execution structure \( E = (E, \rightarrow, \rightarrow') \) is causally atomic w.r.t. \( S \) iff there exists a causally atomic \( S \)-execution structure \( E' = (E', \rightarrow', \rightarrow') \) and an (order-preserving) isomorphism \( \varphi : E \to E' \) such that:

- for each event \( e = (g, p, i) \in E \), where \( i \in I \), we have \( \varphi(e) = (g, p, a) \) for some \( a \in C(i) \), and
- for each event \( e = (g, p, a) \in E \), where \( a \in \Sigma \), we have \( \varphi(e) = (g, p, a) \).

Theorem 10 extends directly to the case of incomplete histories. The fact that each individual object history is causally atomic implies that we can assume the existence of a valid extension for each incomplete event, which is itself causally atomic. Thus, we can apply these per-object extensions to the object-family execution, and show, using the proof of Theorem 10, that the resulting complete execution structure is causally atomic.

18