Making Linearizability Compositional for Partially Ordered Executions

Simon Doherty[1], Brijesh Dongol[2(✉)], Heike Wehrheim[3], and John Derrick[1]

[1] University of Sheffield, Sheffield, UK
[2] University of Surrey, Guildford, UK
b.dongol@surrey.ac.uk
[3] University of Paderborn, Paderborn, Germany

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 the weak-memory models that describe multicore memory systems. In this paper, we present a generalisation of linearizability 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 orders its events. We apply our proof technique and show correctness of the Treiber stack that blocks on empty, annotated with C11 release-acquire synchronisation.

1 Introduction

Linearizability [23,24] is a well-studied [16] condition that defines correctness of a concurrent object in terms of a sequential specification. It ensures that for each history (i.e., execution trace) 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) [31]) to guarantee contextual refinement [20] and compositionality [24]. Unfortunately, most modern execution environments can only guarantee a partial order of memory events, e.g., due to the effects of relaxed memory [3,5,8,34]. 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 [18]. In this paper, we propose a generalisation of linearizability to cope with partially ordered executions, which we show satisfies compositionality.

C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 110–129, 2018.
https://doi.org/10.1007/978-3-319-98938-9_7

Init: $x = 0, y = 0$

Process 1	Process 2
$1 : x := 1;$	$1 :$ `if`$(y = 1)$
$2 : y := 1;$	$2 :$ `assert`$(x = 1);$

Fig. 1. Writing to shared variables

Init: $\mathrm{S} = \langle\rangle, \mathrm{S}' = \langle\rangle$

Process 1	Process 2
$1 :$ S.Push(1);	$1 :$ `if`(S'.Pop $= 1$)
$2 :$ S'.Push(1);	$2 :$ `assert`(S.Pop $= 1$);

Fig. 2. Writing to shared stacks

To motivate the problem consider the following. Figures 1 and 2 show two examples[1] of multi-threaded programs on which weak memory model effects can be observed. Figure 1 shows two threads writing to and reading from two shared variables x and y. Under SC, the `assert` in process 2 never fails: if y equals 1, x must also equal 1. However, this is not true in weak memory models like C11 [8,28]: 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 [8,28]. 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 `assert` will never fail since linearizable objects are expected to be *compositional* [23,24], 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 `assert` fails, i.e., the composition of the two stacks is not linearizable. The issue here is that linearizability, when naively applied to a weak memory setting, allows too many operations to be considered "unordered".

Failure of compositionality is repaired by strengthening the requirements of linearizability on partial orders. Namely, we require the ability to infer enough order in an execution to ensure that the method call S.Push(1) *precedes* S.Pop, forcing S.Pop to return 1, whenever S'.Push(1) precedes S'.Pop.

The contributions of this paper are as follows.

- Our main contribution is the development of a new compositional notion of correctness; we call this condition *causal linearizability*.
- We establish two meta-theoretical properties of causal linearizability. First, we show that, as expected, causal linearizability reduces to linearizability when the underlying memory model is totally ordered. Second, we show that that causal linearizability is the weakest possible strengthening of linearizability that guarantees compositionality, i.e., any correctness condition stronger than linearizability that is also compositional must imply causal linearizability.
- We present 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

[1] Example in Fig. 2 inspired by H.-J. Boehm talk at Dagstuhl, Nov. 2017.

that enables refinement-based verification, building on existing techniques for linearizability in SC [13,16,36].
– We apply this proof technique and verify causal linearizability of a blocking version of the Treiber Stack executing in the C11 weak memory model. For the standard Treiber Stack under C11, we identify a synchronisation pitfall when using only release-acquire synchronisation.

Causal linearizability is so called because it takes into account the causal relationship between events in a way that is relevant to weak memory models. There is an earlier definition of a condition also called "causal linearizability" introduced by Doherty and Derrick in [11]. However, this earlier definition considers causality at the level of (interleaved) sequences and only applies to memory models such as TSO, that satisfy certain operational properties.[2] In contrast, the definition in this paper (Definition 6) considers causality directly over partial orders, making it applicable to a wider range of memory models.

The definition of causal linearizability in this paper is built on the same concerns as the earlier definition in [11], but is not a generalisation of it in a technical sense. Thus Definition 6 does not reduce to the condition in [11], or vice versa, although both reduce to classical linearizability [24]. All mentions of "causal linearizability" in this paper refers to Definition 6. Further comparisons to related correctness conditions are given in Sect. 8.

Causal linearizability is defined in terms of an *execution structure* [32], taking two different relations over operations into account: a "precedence order" (describing operations that are ordered in real time) and a "communication relation". Execution structures allow one to infer the additional precedence orders from communication orders (see Definition 3 (A5)). Applied to Fig. 2, for a weak memory execution in which the `assert` fails, the execution restricted to stack S would not be causally linearizable in the first place (see Sect. 3 for full details). Execution structures are generic, and can be constructed for any weak memory execution that includes method invocation/response events. We develop one such scheme for mapping executions to execution structures based on the *happens-before* relation of the C11 memory model.

This paper is structured as follows. We present our motivating example, the Treiber Stack in C11 in Sect. 2; describe the problem of compositionality and motivate our execution-structure based solution in Sect. 3; and formalise causal linearizability and compositionality in Sect. 4. Causal linearizability for C11 is presented in Sect. 5, and verification of the stack described in Sect. 6. Section 7 describes a synchronisation pitfall.

2 Treiber Stack in C11

The example we consider (see Algorithm 1) is the Treiber Stack [40] (well-studied in a SC setting, but not in a weak memory one), executing in a recent version of the C11 [30] memory model. In C11, commands may be annotated, e.g., R

[2] In retrospect, the name "causal linearizability" is more fitting to this current paper.

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^R(&Top, top, n)
10: function POP
11:     repeat
12:         repeat
13:             top :=^A Top ;
14:         until top ≠ null ;
15:         ntop := top.nxt ;
16:     until CAS^R(&Top, top, ntop)
17:     return top.val ;
```

(for release) and A (for acquire), which introduces extra synchronisation, i.e., additional order over memory events [8,28]. 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 [8]. Full details of the C11 memory model are deferred until Sect. 5.

Due to weak memory effects, the events under consideration, including method invocation and response events are partially ordered [5,6,14,28,30]. As we show in Sect. 3, it turns out that one cannot simply reapply the standard notion of linearizability in this weaker setting; compositionality demands that we use modified correctness condition, causal linearizability, that additionally requires "communication" across conflicting operations.

In Algorithm 1, all accesses to 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[3] causal linearizability of the Stack (see Sect. 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 Sect. 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* [32].

[3] 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 Sect. 6 for details).

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* iff it is reflexive, antisymmetric and transitive, and a *strict order*, iff it is irreflexive, antisymmetric and transitive. A partial or strict order r is a *total order on* Xiff either $(a, b) \in r$ or $(b, a) \in r$ for all $a, b \in X$. We typically use notation such as $<$, $\prec$, $\longrightarrow$ to denote orders, and write, for example, $a < b$ instead of $(a, b) \in <$. We let X^* denote the set of all finite sequences over X, let $\langle\rangle$ denote the empty sequence and use $\circ$ as a concatenation operator on sequences. For a sequence w, we let $\twoheadrightarrow_w$ be the (total) order on its elements: $e \twoheadrightarrow_w e'$ if $w = w_1 \circ \langle e \rangle \circ w_2 \circ \langle e' \rangle \circ w_3$.

Fix a set of *invocations* Inv and a set of *responses* Res. A pair from $Inv \times Res$ represents an *operation*. Each invocation includes both a *method name* and any arguments passed to the method; each response includes any values returned from the method. For example, for a stack S of natural numbers, the *invocations* of the stack might be represented by the set $\{\textsc{Push}(n) \mid n \in \mathbb{N}\} \cup \{\textsc{Pop}\}$, and the *responses* by $\mathbb{N} \cup \{\bot, \texttt{empty}\}$, and the set of operations of the stack is

$$\Sigma_{\mathrm{S}} = \{(\textsc{Push}(n), \bot), (\textsc{Pop}, n) \mid n \in \mathbb{N}\} \cup \{(\textsc{Pop}, \texttt{empty})\}.$$

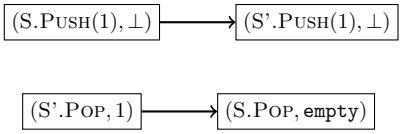
In an execution, an occurrence of an invocation, response, or operation will take the form of an *event*. In a full treatment, events would have the form $e = (l, t, g)$, where l is a *label* of type $Inv \cup Res$ (for executions of a concrete implementation) or $Inv \times Res$ (for executions of an abstract specification), t is a *thread identifier* t from some given set of threads (or processes) Tid and g is a *tag* uniquely identifying the event in the execution. However, for clarity of presentation, we omit tags in this paper. Furthermore, for uniformity, we assume that all invocations, responses and operations are indexed by thread identifiers. For example, the invocations are now given by the set $\{\textsc{Push}_t(n), \textsc{Pop}_t \mid t \in T \wedge n \in \mathbb{N}\}$. We only make thread ids explicit when necessary. We let $tid(e)$ be the thread identifier of event e. For a sequence or partial order of events w, we let $w\lfloor_t$ be the restriction of w to events e with $tid(e) = t$ only.

The standard notion of linearizability is defined for a concurrent history, which is a sequence (or total order) of *invocation* and *response* events of operations. Since operations are concurrent, an invocation of an operation may not be directly followed by its matching response in this sequence, and hence, a history induces a partial order on operations (through the total order on events). For linearizability, we focus on the *precedence* order (denoted $\longrightarrow$), where, for operations o and o', we say $o \longrightarrow o'$ in a history iff the response of operation o occurs before the invocation of operation o' in the history. A concurrent implementation of an object is linearizable if the precedence order ($\longrightarrow$) for *any* history of the object can be extended to a total order that is *legal* for the object's specification [24]. It turns out that linearizability in this setting is *compositional* [23,24]: any history of a family of linearizable objects is itself guaranteed to be linearizable.

Unfortunately, histories in modern execution 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 *compositionality*. To see this, consider the following example.

Example 1. Consider an execution of Fig. 2 where the operations are only ordered by a *happens-before* relation, which is a relation present in many weak-memory models [3,5,8,34]. Since we do not have a global notion of time, we say operation o precedes o' (denoted $o \longrightarrow o'$) if the response of o happens before the invocation of o' (also see [18]). For the C11 memory model, happens-before includes program order, and hence, the progam in Fig. 2 may generate the following execution, where operations executed by the same thread are precedence ordered.

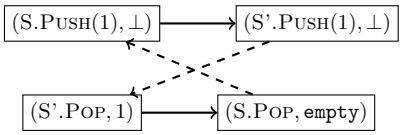


If we restrict the execution above to S only, we can obtain a legal stack behaviour by linearizing (S.Pop, `empty`) before (S.Push(1), ⊥) without contradicting the precedence order $\longrightarrow$ in the diagram above. Similarly, the execution when restricted to S′ is linearizable. However, the full execution is not linearizable: ordering the pop of S before its push, and the push of S' before its pop contradicts the precedence order $\longrightarrow$. □

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

Our first insight is that one must augment the precedence order with additional information about the underlying concurrent execution. In particular, one must introduce information about the *communication* between operations, e.g., when one operation sees the effects of another one. In our example, a pop would see the effect of a push; in the Treiber algorithm it would specifically see the change of Top. Causal linearizability states that the ordering we impose during linearization has to (a) preserve the precedence order of operations and (b) has to be consistent with the communication order. We represent communication by a relation $\dashrightarrow$.

Example 2. Consider again the partial order in Example 1. For stack S, we must linearize pop before push, and for stack S', push before pop. Causal linearizability mandates the existence of a *logical order* that contains $\longrightarrow$ such that all linear extensions of the logical order are legal w.r.t. the specification object. Moreover, it requires that this logical order is contained within the communication relation. Hence, in Example 1, neither S nor S' is causally linearizable: for S, the only valid logical order is S.Pop before S.Push(1), but there is currently no communication from S.Pop to S.Push(1). Thus, the execution in Example 1 is not a counterexample to compositionality of causal linearizability. Now consider changing the example by introducing communication as follows:



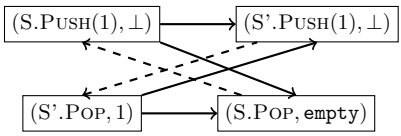
Here, communication is introduced in a manner consistent with the logical order, which requires that (S.Pop, `empty`) is linearized before (S.Push(1), ⊥) and that (S'.Push(1), ⊥) is linearized before (S'.Pop, 1). So far, we would consider this to be a valid counterexample to compositionality. We describe why this cannot be the case below. □

Our second insight is that the operations (taken as events) together with the precedence order $\longrightarrow$ and the communication relation $\dashrightarrow$ must form an *execution structure* [32].

Definition 3 (Execution structure). *If E is a finite*[4] *set of events, and $\longrightarrow, \dashrightarrow \subseteq E \times E$ are relations over E (the* precedence order *and* communication relation*), an* execution structure *is a tuple $(E, \longrightarrow, \dashrightarrow)$ satisfying the following axioms for $e_1, e_2, e_3 \in E$.*

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

Example 4. We apply Definition 3 to Example 2. The requirements of an execution structure, in particular axiom **A4** necessitate that we introduce additional precedence order edges $\longrightarrow$ as follows.



For example, the edge (S'.Pop, 1)$\longrightarrow$(S'.Push(1), ⊥) is induced by the combination of edges (S.Push(1), ⊥)$\longrightarrow$(S'.Push(1), ⊥)$\dashrightarrow$(S'.Pop, 1)$\longrightarrow$(S.Pop, `empty`) together with axiom **A4**. However, this structure now fails to satisfy axiom **A2** and is thus no longer a proper execution structure. □

Hence, for our running example, compositionality no longer fails. We conclude that for causally linearizable stacks, the `assert` in Fig. 2 always holds if it is executed.

[4] The original presentation allows infinite execution structures but requires that $\longrightarrow$ be well founded.

4 Causal Linearizability

Causal linearizability extends linearizability to cope with partially ordered executions. Next, we will formally define it and its compositionality property.

Like ordinary linearizability, causal linearizability is defined by comparing the behaviour of concurrent executions to *legal* sequential ones. The comparison typically proceeds by bringing concurrent operations in sequence under some given constraints. This basic principle is kept for the partially ordered case. Legality is defined by a *sequential object* specification, which we define operationally.

Definition 5. *A* sequential object *is a 4-tuple* $(\Sigma, \Gamma, init, \tau)$*, where* Σ *is a set of* labels*,* Γ *is a set of* states*,* $init \in \Gamma$ *is an* initial state*, and* $\tau \subseteq \Gamma \times \Sigma \times \Gamma$ *is a* transition relation*.*

The set $\Sigma \subseteq Inv \times Res$ consists of pairs of invocations and responses. For our stack example, $\Gamma = \mathbb{N}^*$, $init = \langle\rangle$ and

$$\begin{aligned}\tau = &\{(s, (\textsc{Push}(n), \bot), \langle n\rangle \circ s) \mid n \in \mathbb{N}\} \cup \{(\langle n\rangle \circ s, (\textsc{Pop}, n), s) \mid n \in \mathbb{N}\} \\ &\cup \{(\langle\rangle, (\textsc{Pop}, \texttt{empty}), \langle\rangle)\}\end{aligned}$$

We write $s \overset{op}{\rightsquigarrow}_\tau s'$ for $(s, op, s') \in \tau$. For a sequence $w \in \Sigma^*$, we write $s \overset{w}{\rightsquigarrow}_\tau s'$ iff either $w = \langle\rangle$ and $s = s'$, or $w = \langle op\rangle \circ w'$ and there exists an s'' such that $s \overset{op}{\rightsquigarrow}_\tau s''$ and $s'' \overset{w'}{\rightsquigarrow}_\tau s'$. The set of *legal histories* of an object $\mathbb{S} = (\Sigma, \Gamma, init, \tau)$ is given by $legal_{\mathbb{S}} = \{w \in \Sigma^* \mid \exists s \in \Gamma.\ init \overset{w}{\rightsquigarrow}_\tau s\}$.

In general, executions of concurrent processes might invoke operations on more than one object. To capture this, we define a notion of an *object product*. If $\mathbb{S}_1 = (\Sigma_1, \Gamma_1, init_1, \tau_1)$ and $\mathbb{S}_2 = (\Sigma_2, \Gamma_2, init_2, \tau_2)$ are two sequential objects with $\Sigma_1 \cap \Sigma_2 = \emptyset$, the object product of $\mathbb{S}_1$ and $\mathbb{S}_2$ is defined by $\mathbb{S}_1 \otimes \mathbb{S}_2 = (\Sigma_1 \cup \Sigma_2, \Gamma_1 \times \Gamma_2, (init_1, init_2), \tau_1 \otimes \tau_2)$, where

$$\begin{aligned}\tau_1 \otimes \tau_2 = &\{((s_1, s_2), op_1, (s_1', s_2)) \mid op_1 \in \Sigma_1 \wedge (s_1, op_1, s_1') \in \tau_1\} \\ &\cup \{((s_1, s_2), op_2, (s_1, s_2')) \mid op_2 \in \Sigma_2 \wedge (s_2, op_2, s_2') \in \tau_2\}.\end{aligned}$$

Clearly, this construction can be generalised to products of more than two objects, provided their sets of actions are pairwise disjoint. We abstain from such a treatment here since a compositionality result for two objects is sufficient to ensure compositionality over multiple objects.

Causal linearizabilty compares the concurrent execution given by an execution structure to the legal sequential behaviour. The constraint on this sequentialization is that the precedence and the communication order of execution structures provide lower and upper bounds for the allowed ordering. More precisely, we use a partial order $<$ that contains all orders necessary to ensure legality, i.e., there is no linear extension of $<$ that is not legal. Causal linearizability requires (a) the precedence order of execution structures to be preserved by this order, and (b) this order to be contained in the communication relation. We say a strict partial order $<$ is a *logical order* of an execution structure

$\mathbb{E} = (E, \longrightarrow, \dashrightarrow)$ iff $< \subseteq E \times E$ and $\longrightarrow \subseteq < \subseteq \dashrightarrow$. For concurrent objects, one possible instantiation of a logical order is given in [12], where the logical order corresponds to a conflict-based notion of causality.

For a partial order $< \subseteq E \times E$, we let $LE(<) = \{w \in E^* \mid < \subseteq \twoheadrightarrow_w\}$ be the set of *linear extensions* of $<$.

Definition 6. *Let $\mathbb{S}$ be a sequential object. An execution structure $\mathbb{E}$ is* causally linearizable *w.r.t. $\mathbb{S}$ iff there exists a logical order $<$ of $\mathbb{E}$ such that $LE(<) \subseteq legal_{\mathbb{S}}$.*

Causal linearizability guarantees compositionality, i.e., the composition of causally linearizable concurrent objects is causally linearizable. More formally, for an execution structure $\mathbb{E} = (E, \longrightarrow, \dashrightarrow)$ and events $X \subseteq E$, we let $\mathbb{E}\lfloor_X$ be the execution structure restricted to X, i.e., $(X, \longrightarrow \cap (X \times X), \dashrightarrow \cap (X \times X))$.

Theorem 7 (Compositionality). *If $\mathbb{S}_1 = (\Sigma_1, \ldots)$ and $\mathbb{S}_2 = (\Sigma_2, \ldots)$ are sequential objects with $\Sigma_1 \cap \Sigma_2 = \emptyset$ and $\mathbb{E} = (E, \longrightarrow, \dashrightarrow)$ is an execution structure, then $\mathbb{E}\lfloor_{\Sigma_1}$ is causally linearizable w.r.t. $\mathbb{S}_1$ and $\mathbb{E}\lfloor_{\Sigma_2}$ causally linearizable w.r.t. $\mathbb{S}_2$ iff $\mathbb{E}$ is causally linearizable w.r.t. $\mathbb{S}_1 \otimes \mathbb{S}_2$.* □

Standard linearizability as introduced by Herlihy and Wing [24] is defined on executions (histories) which are totally ordered sequences of invocations and responses of operations, i.e. a history h is an element of $(Inv \cup Res)^*$. Note that this allows executions in which operations are concurrent because invocation and response events are now separated. Histories are required to be *well-formed* and *complete*[5], which means that the projection of a history onto one thread forms a sequence of invocations and corresponding responses.

A strict order $\prec$ on $Inv \cup Res$ is well-formed and complete iff for all threads $t \in Tid$, $\prec\lfloor_t$ is *sequential*, i.e., forms an alternating total order of invocations and responses starting with an invocation. Invocations and responses thus form *matching pairs* as defined by a function mp. For a strict order $\prec$ such that $i \prec r$, $(i, r) \in mp(\prec)$ iff $tid(i) = tid(r)$ and there is no event e such that $i \prec e \prec r$ and $tid(e) = tid(i)$. This allows us to derive an execution structure from any strict order and thus also from a history h by using its ordering $\twoheadrightarrow_h$.

Definition 8. *Let $\prec$ be a well-formed and complete strict order on $Inv \cup Res$. We say $exec(\prec) = (E, \longrightarrow, \dashrightarrow)$ is the execution structure corresponding to $\prec$ if*

$$\begin{aligned} E &= mp(\prec), \\ \longrightarrow &= \{((i_1, r_1), (i_2, r_2)) \in E \times E \mid r_1 \prec i_2\}, \\ \dashrightarrow &= \{((i_1, r_1), (i_2, r_2)) \in E \times E \mid i_1 \prec r_2\}. \end{aligned}$$

Note that this construction guarantees a *saturation* property: for two events e, e', we either have $e \longrightarrow e'$ or $e' \dashrightarrow e$.

The classical definition of linearizability only employs the precedence ordering of an execution structure. That is, $(E, \longrightarrow, \dashrightarrow)$ is linearizable w.r.t. a sequential

[5] Note that we only assume completeness for the sake of simplicity here.

object $\mathbb{S}$ iff there exists a sequence $hs \in \mathit{legal}_{\mathbb{S}}$ such that (i) $\longrightarrow\restriction_t = \twoheadrightarrow_{hs}\restriction_t$ for all $t \in \mathit{Tid}$ (threads execute the same sequence of operations) and (ii) $\longrightarrow \subseteq \twoheadrightarrow_{hs}$ (precedence ordering between operations is preserved). We say that a strict order $\prec$ is linearizable iff $\mathit{exec}(\prec)$ is linearizable and that a history h is linearizable iff $\twoheadrightarrow_h$ is linearizable.

Theorem 9. *Suppose h is a history and $\mathbb{S}$ a sequential object. Then h is linearizable w.r.t. $\mathbb{S}$ iff $\mathit{exec}(\twoheadrightarrow_h)$ is causally linearizable w.r.t. $\mathbb{S}$.* □

We now provide an adequacy result for causal linearizability, i.e., show that causal linearizability is, in fact, the *weakest* possible strengthening of linearizability that is compositional. The technical exposition is formalised in terms of correctness conditions that guarantee linearizability. Here, we regard a correctness condition to be a function mapping a sequential object to the set of all well-formed strict orders on $\mathit{Inv} \cup \mathit{Res}$ accepted as being correct for the object, where the mapping is closed under renaming of the operations.

We let $\mathcal{S}$ be the set of all possible sequential objects and $\mathcal{H}$ the set of all possible well-formed complete strict orders on $\mathit{Inv} \cup \mathit{Res}$. To formalise closure under renaming, we assume a bijection $b : X \to Y$ between sets X and Y. If $\mathbb{S} = (X, \Gamma, \mathit{init}, \tau)$ is a sequential object, define $b[\mathbb{S}] = (Y, \Gamma, \mathit{init}, b[\tau])$, where $b[\tau] = \{(s, b(x), s') \mid (s, x, s') \in \tau\}$ and if $w \in X^*$, define $b[w] \in Y^*$ to be the sequence obtained from w by replacing each w_i by $b(w_i)$.

Definition 10. *We say a function $\Delta : \mathcal{S} \to 2^{\mathcal{H}}$ is a* correctness condition *iff Δ is closed under renaming of operations, i.e., for all bijective functions $b : \Sigma \to \Sigma'$ and for all $\mathbb{S} = (\Sigma, \dots) \in \mathcal{S}$, we have $\prec \in \Delta(\mathbb{S})$ iff $b[\prec] \in \Delta(b[\mathbb{S}])$.*

Definition 11. *We say $\Delta : \mathcal{S} \to 2^{\mathcal{H}}$* guarantees linearizability *iff for all $\mathbb{S} \in \mathcal{S}$, each $\prec \in \Delta(\mathbb{S})$ is linearizable w.r.t. $\mathbb{S}$.*

Our adequacy result for causal linearizability is defined for well-formed strict orders that have exactly one possible legal linearization. Formally, for a correctness condition Δ and sequential object $\mathbb{S}$, we say $\prec \in \Delta(\mathbb{S})$ is *strongly synchronised* iff it is linearizable w.r.t. exactly one $hs \in \mathit{legal}_{\mathbb{S}}$.

Theorem 12. *Let Δ be a compositional correctness condition, and let $\mathbb{S}$ be a sequential object. Then for any strongly synchronised strict order $\prec \in \Delta(\mathbb{S})$, $\mathit{exec}(\prec)$ is causally linearizable.*

Strong synchronisation may always arise for some specifications, e.g., a data structure such as a stack or a counter object that only provides a *fetch-and-increment* operation. In general, the execution of any object may be strongly synchronised due to interactions with other objects or a client (see [18]), causing additional precedence order to be introduced. For example, a client thread of a concurrent object may introduce precedence order via program order, inserting fences between operation calls, or calling objects that induce additional order [6, 18]. Thus for typical sequential objects, a correctness condition that prohibited strongly synchronized executions would be overly restrictive. Theorem 12 ensures that, for such executions, if the correctness condition is compositional then it is at least as strong as causal linearizability.

5 C11 Executions

We now briefly introduce the C11 memory model as to be able to reason about programs executing within C11. To this end, we simply give a (condensed) adaption of the programming-language oriented presentation of C11 [14,28], but we ignore various features of C11 not needed for our discussion, including non-atomic operations and fences. For a more complete explanation see e.g. [28].

The C11 Memory-Model. The memory model specifies how read and write operations access shared state. Let L be a set of such shared *locations* (ranged over by x, y) and 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* (read-modify-write) events, U. A read event would e.g. take the form $rd(x, 0)$. An update event occurs for instance when a CAS operation is executed: a shared location is read, compared to a local variable and then possibly written to. We 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 accessed location and $Loc(x) = \{e \mid loc(e) = x\}$ the set of events accessing location 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 (in our restricted setting) may either be a release, R, or an acquire, A, annotation, and we let $ann(e)$ be an event e's annotation.

Definition 13. *A* C11 execution *is a tuple* $\mathbb{D} = (D, sb, rf, mo)$*, where* D *is a set of events, and* $sb, rf, mo \subseteq D \times D$ *define the* sequenced-before*,* reads-from *and* modification order *relations, respectively.*

We say a C11 execution is *valid* when it satisfies the following constraints: **(V1)** sb is a strict order, such that, for each process p, the projection of sb onto p is a total order; the reads-from relation specifies the write a particular read event reads-from: $rf \subseteq Mod \times Qry$ and **(V2)** for all $(w, r) \in rf$, $loc(w) = loc(r)$ and $wval(w) = rval(r)$ as well as **(V3)** for all $r \in D \cap Qry$, there exists some $w \in D \cap Mod$ such that $(w, r) \in rf$; the modification order relates writes to the same location and these writes are totally ordered: **(V4)** for all $(w, w') \in mo$, $loc(w) = loc(w')$; and **(V5)** for all $w, w' \in Mod$ 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, $sw = rf \cap (D_{\mathsf{R}} \times D_{\mathsf{A}})$, creates interthread ordering guarantees based on synchronisation annotations. The annotations R and A can thus be used by programmers to achieve certain visibility effects of their write events. The *from-read* relation, $fr = (rf^{-1}; mo) \backslash Id$, 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 memory model are low level read and write events. 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 [18], program order in C11 is consistent with happens-before order, and hence, invocation and response events 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. In any execution of a well-formed program, this condition is naturally satisfied.

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 $\mathbb{D}$ and $IR = Inv \cup Res$, we let $hb_{ir} = hb \cap (IR \times IR)$, i.e., the happens-before relation of $\mathbb{D}$ restricted to the invocation and response events. By **(V6)**, hb_{ir} is well-formed and complete. Thus, we can apply the construction defined in Sect. 4 to build an execution structure $exec(hb_{ir})$.

Definition 14. *We say that a C11 execution* $\mathbb{D}$ *is* causally linearizable *w.r.t. a sequential object if* $exec(hb_{ir})$ *is.* □

Compositionality of causal linearizability thus naturally carries over to C11 executions. Finally, we say that a data structure (like the Treiber stack) is causally linearizable on C11 when all its C11 executions are. Thus, we will in the following investigate how we can prove such a property.

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. Our method is based on a simulation-based proof rule described in Sect. 6.1. We illustrate our technique on the Treiber Stack (Sect. 6.2), which is often used as an exercise in the verification literature [16]. Unlike these existing verifications, we consider weak memory, and hence, the stack in Algorithm 1 generates more behaviours than in a standard sequentially consistent setting. The proof in Sect. 6.2 below is the first to verify that the stack under C11 satisfies causal linearizability. Moreover, our proof technique, which considers simulation over a happens-before relation, is novel to this paper.

6.1 A Simulation Relation over Happens-Before

For the remainder of this section, fix a C11 execution $\mathbb{D} = (D, sb, rf, mo)$, and a sequential object $\mathbb{S} = (\Sigma, \Gamma, init, \tau)$. We describe a method for proving that

$\mathbb{D}$ is causally linearizable w.r.t. $\mathbb{S}$. In what follows, we write $e \longrightarrow_{hb} e'$ when $(e, e') \in hb_{\mathbb{D}}$.

As in the interleaving setting our method depends on assigining *linearization points* [16] to each operation. Therefore, the verifier must define a function $lp : D \cap Inv \rightarrow D$, which returns the memory event that linearizes the given high-level operation, represented by its invocation. For simplicity, in this presentation, we require that our linearizations be injective.[6] Recall from the previous section that the operations in the execution structure $exec(hb_{ir})$ are elements of matching pairs from the set $mp(hb_{ir})$. To recover the abstract order of operations corresponding to a linearization, we use $<_{lp} = \{((i, r), (i', r')) \mid lp(i) \longrightarrow_{hb} lp(i') \wedge (i, r), (i', r') \in mp(hb_{ir})\}$.

Definition 15. *We say lp is a* linearization *iff for each $i \in D \cap Inv$, and matching response r, $i \longrightarrow_{hb} lp(a) \longrightarrow_{hb} r$. Furthermore, we say lp is a* legal linearization *iff $LE(<_{lp}) \subseteq legal_{\mathbb{S}}$.*

Note that, for a legal linearization, we require that *every* linear extension of $<_{lp}$ yields a legal history under the linearization function lp. Of course, if $\longrightarrow_{hb}$ were total, this would reduce to essentially the standard notion of linearization point, and thus our proof technique is a generalization of a standard technique.

The existence of a legal linearization is sufficient to prove causal linearizability of the C11 execution.

Theorem 16 (Legal linearizations guarantee causal linearizability). *If there is a legal linearization lp, then D is causally linearizable w.r.t. $\mathbb{S}$.*

The key difficulty in using Theorem 16 is showing that a given linearization function is legal. To this end, we extend the standard simulation method to prove legality of a linearization function [16].

In the usual setting, a simulation relation relates states of the implementation to states of the specificiation, and this relation encodes an induction hypothesis for an induction on the executions of the specification. In our current setting, the simulation relation (which we denote ρ, below) relates *sets* of low-level actions to abstract states. The simulation relation encodes an induction hypothesis for an induction on the linear extensions of the *hb*-relation. Thus, at each stage of the induction we can assume $\rho(Z, \gamma)$ for some set of events $Z \subseteq D$ and state $\gamma \in \Gamma$, where Z is downwards-closed with respect to the *hb* order. The set Z is the set of low-level actions already considered by the induction. In order to be a simulation, the relation ρ must satisfy the conditions given in the following definition.

Definition 17 (*hb*-simulation). *Suppose lp is a linearization. An hb-*simulation *is a relation $\rho \subseteq 2^D \times \Gamma$ such that:*

1. $\rho(\emptyset, init)$, *and* *(initialisation)*
2. *for all $Z \subseteq D$, events $e \in D \backslash Z$ and $\gamma \in \Gamma$, if $\forall e' \in D.\ e' \longrightarrow_{hb} e \Rightarrow e' \in Z$ and $\rho(Z, \gamma)$ then*

[6] Thus, each low-level event can linearize at most one action of the specification.

(a) *if* $e \notin \mathbf{ran}\, lp$, *then* $\rho(Z \cup \{e\}, \gamma)$, *and* *(stutter step)*
(b) *if* $e = lp(i)$ *for some* $i \in D \cap Inv$,
then, letting r *be the matching response of* i *in* D, $(\gamma, (i, r), \gamma') \in \tau$ *and* $\rho(Z \cup \{e\}, \gamma')$ *for some* $\gamma' \in \Gamma$. *(linearization step)*

Condition 1 ensures that the initial states match up: at the concrete level this is the empty set and at the abstract level, this is the initial state. The induction considers the low-level actions in *hb* order, the low-level action under consideration must be an element of $D \backslash Z$ such that all its *hb* predecessors are already in Z. For each such event e, there are two possibilities: either e is a stutter step (in which case the abstract state is unchanged), or e linearizes the operation invoked by i (in which case the abstract system takes a step). In either case, the event e is added to the set Z, and we must show that the simulation relation is preserved.

The existence of an *hb*-simulation guarantees that *lp* is a legal linearization. This fact is captured by the next theorem.

Theorem 18 (*hb*-simulation guarantees legal linearization). *If lp is a linearization and* ρ *is an hb-simulation with respect to lp, then lp is a legal linearization.*

Thus, if we can exhibit a linearization *lp* and an *hb*-simulation ρ, then $\mathbb{D}$ is causally linearizability w.r.t. $\mathbb{S}$.

6.2 Case Study: The Treiber Stack

We now describe a linearization function *lp* and an *hb*-simulation relation ρ, demonstrating causal linearizability of the Treiber stack. We fix some arbitrary C11 execution $\mathbb{D} = (D, sb, rf, mo)$ that contains an instance of the Treiber stack. That is, the invocations in $\mathbb{D}$ are the stack invocations, and the responses are the stack responses (as given in Sect. 3). Furthermore, the low-level memory operations between these invocations and responses are generated by executions of the operations of the Treiber stack (Algorithm 1). As before, we write $e \longrightarrow_{hb} e'$ when $(e, e') \in hb_{\mathbb{D}}$.

The linearization function *lp* for the Treiber stack is completely standard: referring to Algorithm 1 on page 4, 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).

The main component of our simulation relation ρ 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

$$latest_Z(x) = max(mo \downarrow_{(Z \cap Loc(x))})$$

and the *current value* of a location x in some set Z as $cval_Z(x) = wval(latest_Z(x))$, 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 $stackOf_Z(x) = v \cdot stackOf_Z(y)$, where $v = cval_Z(x.val)$ and $y = cval_Z(x.nxt)$.

Now, assuming that s is a state of the sequential stack, our simulation requires:

$$stackOf_Z(cval_Z(\&Top)) = s \tag{1}$$

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

$$\forall m, m' \in Z \cap Mod(\&Top).\ m \longrightarrow_{hb} m' \vee m' \longrightarrow_{hb} m \tag{2}$$

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

In what follows, we illustrate how to verify the proof obligations given in Definition 17, 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,

$$\forall m \in Z \cap Mod \cap Loc(\&Top).\ m \longrightarrow_{hb} e \tag{3}$$

Proving this formally is somewhat involved, but the essential reason is as follows. Note that there is an acquiring read r to $\&Top$ executed at line 7 of e's operation and sb-prior to e. r reads from some releasing update u. Thus, by Property 2, and the fact the hb contains sb, e is happens after u, and all prior updates. If there were some update u' of $\&Top$ such that $(u', e) \notin hb$, then $(u', u) \notin hb$ so by Property 2, $u \longrightarrow_{hb} u'$. 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 3 makes it straightforward to verify that Condition 2b of Definition 17 is satisfied. To see this, note that every linearization point of every operation is a modification of $\&\ Top$. Thus, if (i', r') is some operation such that $lp(i') \in Z$ (so that this operation has already been linearized) then $lp(i') \longrightarrow_{hb} e$.

Using Property 3 it is easy to see that both Propertys 1 and 2 are preserved. We show by contradiction that $latest_{Z'}(\&Top) = e$. Otherwise, we have $(e, latest_{Z'}(\&Top)) \in mo$. Therefore $(latest_{Z'}(\&Top), e) \notin hb$, but $latest_{Z'}(\&Top)$ is a modification operation, so this contradicts Property 3.

It follows from $latest_{Z'}(\&Top) = e$ that $stackOf(cval_{Z'}) = stackOf(wval(e))$. Given this, it is straightforward to show that Property 1 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 *val* field of the node being added to the stack (formally, $cval_Z((wval(e)).nxt)$) is the value passed to the push operation; and that the current value of the *nxt* field (formally, $cval_Z((wval(e)).nxt)$) is the current value of $\&Top$ when the successful CAS occurs. These properties can be proved using the model of dynamic memory given in Sect. 5.

7 A Synchronisation Pitfall

We now describe an important observation regarding failure of causal linearizability 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. Note that this observation does not indicate that our definition of causal linearizability is too strong, but rather that release-acquire annotations cannot guarantee the communication from a read-only operation to a writing operation necessary for compositionality.

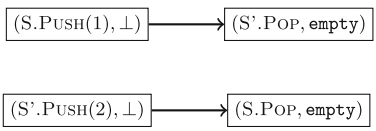


Fig. 3. Read-only operations without communication (not compositional)

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 :=$^{\mathsf{A}}$ 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 Sect. 3, is not compositional. Recovering compositionality requires one to introduce additional communication edges from the pops returning empty to the corresponding push operations. 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 SC annotations, which can induce synchronisation across from-read edges. However, the precise meaning of these annotations is still a topic of active research [7,30].

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 [15,25], 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, it is possible to extend this condition to cover, for example, transactional memory.

Causal linearizability is closely related to *causal hb-linearizability* defined in [18], 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. In previous work (see also Sect. 1), we have defined a correctness condition that is only compositional when the client satisfies certain constraints [11]; in contrast, the treatment in this paper achieves a full decoupling between the client and object. Furthermore, that condition is only defined when the underlying memory model is given operationally, rather than axiomatically like C11. Early attempts, targeting TSO architectures, used totally ordered histories but allowed the response of an operation to be moved to a corresponding "flush" event [10,17,21,39]. 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 [6]. 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 [4] is a related condition, aimed at shared-memory and data-stores, which has no notion of precedence order and is not compositional.

There exists a rich body of work on the semantics of weak memory models, including semantics for the C11 memory model [3,5,7,8,26,30,35]. This has been used as a basis for program logics [14,15,22,25,29,38] and given rise to automated tools for analysis of weak memory programs [1,2,27,37]. These logics and associated verification tools however, are typically not designed to reason about simulation and refinement as is essential for proofs of (causal) linearizability of concurrent data structures [16]. There are several existing automated techniques for checking (classical) linearizability [9,19,33,41] that use simulation-based techniques. We anticipate that such techniques could be extended to verify hb-simulation and causal linearizability, however, leave such extensions as future work.

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. Acta Inform. **54**(8), 789–818 (2017)
2. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: Context-bounded analysis for POWER. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 56–74. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_4
3. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: a tutorial. IEEE Comput. **29**(12), 66–76 (1996)
4. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. Distrib. Comput. **9**(1), 37–49 (1995)
5. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (2014)
6. Batty, M., Dodds, M., Gotsman, A.: Library abstraction for C/C++ concurrency. In: Giacobazzi, R., Cousot, R. (eds.) POPL, pp. 235–248. ACM (2013)
7. Batty, M., Donaldson, A.F., Wickerson, J.: Overhauling SC atomics in C11 and OpenCL. In: POPL, pp. 634–648. ACM (2016)
8. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Ball, T., Sagiv, M. (eds.) POPL, pp. 55–66. ACM (2011)
9. Bouajjani, A., Emmi, M., Enea, C., Mutluergil, S.O.: Proving linearizability using forward simulations. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 542–563. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_28
10. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent library correctness on the TSO memory model. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 87–107. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_5
11. Doherty, S., Derrick, J.: Linearizability and causality. In: De Nicola, R., Kühn, E. (eds.) SEFM 2016. LNCS, vol. 9763, pp. 45–60. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41591-8_4
12. Doherty, S., Derrick, J., Dongol, B., Wehrheim, H.: Causal linearizability: compositionality for partially ordered executions. CoRR abs/1802.01866 (2018)
13. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30232-2_7
14. Doko, M., Vafeiadis, V.: A program logic for C11 memory fences. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 413–430. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_20
15. Doko, M., Vafeiadis, V.: Tackling real-life relaxed concurrency with FSL++. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 448–475. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_17
16. Dongol, B., Derrick, J.: Verifying linearisability: a comparative survey. ACM Comput. Surv. **48**(2), 19:1–19:43 (2015)
17. Dongol, B., Derrick, J., Smith, G.: Reasoning algebraically about refinement on TSO architectures. In: Ciobanu, G., Méry, D. (eds.) ICTAC 2014. LNCS, vol. 8687, pp. 151–168. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10882-7_10

18. Dongol, B., Jagadeesan, R., Riely, J., Armstrong, A.: On abstraction and compositionality for weak-memory linearisability. Verification, Model Checking, and Abstract Interpretation. LNCS, vol. 10747, pp. 183–204. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73721-8_9
19. Doolan, P., Smith, G., Zhang, C., Krishnan, P.: Improving the scalability of automatic linearizability checking in SPIN. In: Duan, Z., Ong, L. (eds.) ICFEM 2017. LNCS, vol. 10610, pp. 105–121. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68690-5_7
20. Filipovic, I., O'Hearn, P.W., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theor. Comput. Sci. **411**(51–52), 4379–4398 (2010)
21. Gotsman, A., Musuvathi, M., Yang, H.: Show no weakness: sequentially consistent specifications of TSO libraries. In: Aguilera, M.K. (ed.) DISC 2012. LNCS, vol. 7611, pp. 31–45. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33651-5_3
22. He, M., Vafeiadis, V., Qin, S., Ferreira, J.F.: Reasoning about fences and relaxed atomics. In: PDP, pp. 520–527. IEEE Computer Society (2016)
23. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, Burlington (2008)
24. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM TOPLAS **12**(3), 463–492 (1990)
25. Kaiser, J., Dang, H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: reasoning about release-acquire consistency in iris. In: ECOOP. LIPIcs, vol. 74, pp. 17:1–17:29. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
26. Kang, J., Hur, C., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: Castagna, G., Gordon, A.D. (eds.) POPL, pp. 175–189. ACM (2017)
27. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. PACMPL **2**(POPL), 17:1–17:32 (2018)
28. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: Bodík, R., Majumdar, R. (eds.) POPL, pp. 649–662. ACM (2016)
29. Lahav, O., Vafeiadis, V.: Owicki-Gries reasoning for weak memory models. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 311–323. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_25
30. Lahav, O., Vafeiadis, V., Kang, J., Hur, C., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI, pp. 618–632. ACM (2017)
31. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9), 690–691 (1979)
32. Lamport, L.: On interprocess communication. Part I: basic formalism. Distrib. Comput. **1**(2), 77–85 (1986)
33. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 321–337. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05089-3_21
34. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL, pp. 378–391. ACM (2005)
35. Nienhuis, K., Memarian, K., Sewell, P.: An operational semantics for C/C++11 concurrency. In: Visser, E., Smaragdakis, Y. (eds.) OOPSLA, pp. 111–128. ACM (2016)
36. Schellhorn, G., Derrick, J., Wehrheim, H.: A sound and complete proof technique for linearizability of concurrent data structures. ACM Trans. Comput. Log. **15**(4), 31:1–31:37 (2014)

37. Summers, A.J., Müller, P.: Automating deductive verification for weak-memory programs. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 190–209. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_11
38. Svendsen, K., Pichon-Pharabod, J., Doko, M., Lahav, O., Vafeiadis, V.: A separation logic for a promising semantics. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 357–384. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_13
39. Travkin, O., Wehrheim, H.: Handling TSO in mechanized linearizability proofs. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 132–147. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13338-6_11
40. Treiber, R.K.: Systems programming: coping with parallelism. Technical report, RJ 5118, IBM Almaden Res. Ctr. (1986)
41. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_40