# Monitoring Weak Consistency

Michael Emmi[1(✉)] and Constantin Enea[2]

[1] SRI International, New York, NY, USA
michael.emmi@sri.com
[2] IRIF, Univ. Paris Diderot and CRNS, Paris, France
cenea@irif.fr

**Abstract.** High-performance implementations of distributed and multicore shared objects often guarantee only the weak consistency of their concurrent operations, foregoing the de-facto yet performance-restrictive consistency criterion of linearizability. While such weak consistency is often vital for achieving performance requirements, practical automation for checking weak-consistency is lacking. In principle, algorithmically checking the consistency of executions according to various weak-consistency criteria is hard: in addition to the enumeration of linearizations of an execution's operations, such criteria generally demand the enumeration of possible visibility relations among the linearized operations; a priori, both enumerations are exponential.

In this work we identify an optimization to weak-consistency checking: rather than enumerating every possible visibility relation, it suffices to consider only the *minimal* visibility relations which adhere to the various constraints of the given criterion, for a significant class of consistency criteria. We demonstrate the soundness of this optimization, and describe an associated minimal-visibility consistency checking algorithm. Empirically, we show that our algorithm significantly outperforms the baseline weak-consistency checking algorithm, which naïvely enumerates all visibilities, and adds only modest overhead to the baseline linearizability checking algorithm, which does not enumerate visibilities.

**Keywords:** Linearizability · Consistency · Runtime verification

## 1 Introduction

Programming software applications that can deal with multiple clients at the same time, and possibly, with clients that connect at different sites in a network, relies on optimized concurrent or distributed objects which encapsulate lock-free shared memory access or message passing protocols into high-level abstract data types. Given the potentially-enormous amount of software that relies on

---

This work is supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 678177).

H. Chockler and G. Weissenbacher (Eds.): CAV 2018, LNCS 10981, pp. 487–506, 2018.
https://doi.org/10.1007/978-3-319-96145-3_26

these objects, it is important to maintain precise specifications and ensure that implementations adhere to their specifications.

One of the standard correctness criteria used in this context is linearizability (or strong consistency) [22], which ensures that the results of concurrently-executed invocations match the results of some serial execution of those same invocations. Ensuring such a criterion in a distributed context (when data is replicated at different sites in a network) is practically infeasible or even impossible [17,19]. Therefore, various weak consistency criteria have been proposed like eventual consistency [23,36], "session guarantees" like read-my-writes or monotonic-reads [35], causal consistency [25,28], etc.

An axiomatic framework for formalizing such criteria has been proposed by Burckhardt et al. [9,11]. Essentially, this extends the linearizability-based specification methodology with a dynamic *visibility* relation among operations, in addition to the standard dynamic *happens-before* and *linearization* relations. Permitting weaker visibility relations models outcomes in which an operation may not observe the effects of concurrent operations that are linearized before it.

In this work, we propose an online monitoring algorithm that checks whether an execution of a concurrent (or distributed) object satisfies a consistency model defined in this axiomatic framework. This algorithm constructs a linearization and visibility relation satisfying the axioms of the consistency model gradually as the execution extends with more operations. It is possible that the linearization and visibility constructed until some point in time are invalidated as more operations get executed, which requires the algorithm to backtrack and search for different candidates. This exponential blow-up is unavoidable since even the problem of checking linearizability is NP-hard in general [18].

The main difficulty in devising such an algorithm is coming up with efficient strategies for enumerating linearizations and visibility relations which minimize the number of candidates needed to be explored and the number of times the algorithm has to backtrack. We build on previous works that propose such strategies for enumerating linearizations [29,38] in the context of linearizability checking. Roughly, the linearizations are extended iteratively by appending operations which are minimal in the happens-before order (among non-linearized operations). The choice of the minimal operations to append varies from one approach to the other. Our work focuses on combining such strategies with an efficient enumeration of visibility relations which are compatible with a given linearization.

Rather than specializing our results to one single consistency model, we consider a general class of consistency models from Burckhardt et al.'s axiomatic framework [9,11] in which the visibility relation among operations is constrained to be contained in the linearization relation. That class includes, for instance, time-stamp based models employed in distributed object implementations, in which time stamps serve to resolve conflicts by effectively linearizing concurrent operations. We show that within this class of consistency models, it is *not* necessary to enumerate the set of all possible visibility relations (included in the linearization) in order to check consistency of an execution. More precisely, we develop an algorithm for enumerating visibility relations that traverses operations in linearization order and chooses for each operation $o$, a *minimal* set of operations visible to $o$ that conforms to the consistency axioms (up to the linearization prefix that includes $o$). In general there may exist multiple such minimal sets of operations, and each of them must be explored. When the visibility relation cannot be extended, the algorithm needs to backtrack and choose different minimal visibility sets for previous operations. However, when all the minimal candidates have been explored, the algorithm can soundly report that the execution is not consistent, without resorting to the exploration of non-minimal visibility relations.

Besides demonstrating the soundness of minimal-visibility consistency checking, we also demonstrate its empirical impact by applying our algorithm to concurrent traces of Java concurrent data structures. We find that our algorithm consistently outperforms the baseline naïve approach to enumerating visibilities, which considers also non-minimal visibility relations. Furthermore, we demonstrate that minimal-visibility checking adds only modest overhead (roughly 2×) to the baseline linearizability checking algorithm, which does not enumerate visibilities. This suggests that small sets of minimal visibilities typically suffice in practice, and that the additional exponential enumeration of visibilities, atop the exponential enumeration of linearizations, may be avoidable in practice. Our implementation and experiments are open source, and publicly available on GitHub.[1]

In summary, this work makes the following contributions:

- we develop a new *minimal-visibility* consistency-checking algorithm for Burckhardt et al.'s axiomatic consistency framework [9,11];
- we demonstrate the soundness of minimal-visibility consistency checking; and
- we demonstrate an empirical evaluation comparing minimal-visibility consistency checking with the state-of-the-art consistency-checking algorithms.

To the best of our knowledge, our algorithm is the first completely automatic algorithm for checking weak-consistency of arbitrary abstract data type implementations which avoids the naïve enumeration of all possible visibility relations.

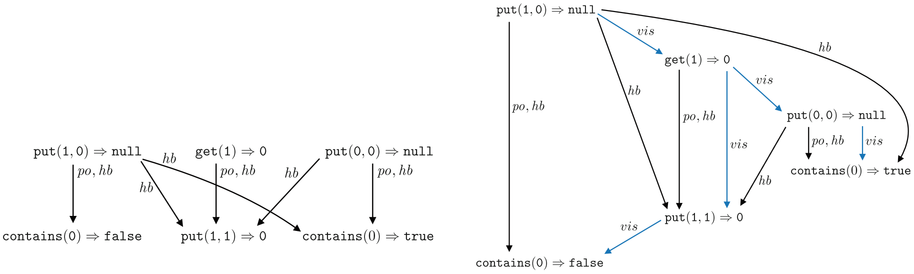
The rest of this paper is organized as follows. Section 2 elaborates a formalization of Burckhardt et al.'s axiomatic consistency framework [9,11], and Sect. 3 develops a formal argument to the soundness of considering only minimal visibility relations. Section 4 describes our overall consistency checking algorithms, and Sect. 5 describes our implementation and empirical evaluation. Section 6 describes related work, and finally Sect. 7 concludes.

## 2 Weak Consistency

We describe a formal model for concurrent (distributed) object implementations. Clients interact with an object by making *invocations* from a set $\mathbb{I}$ and receiving

---

[1] https://github.com/michael-emmi/violat/releases/tag/cav-2018-submission.



(a) (b)

**Fig. 1.** A history $h$ and an abstract execution containing $h$.

*returns* from a set $\mathbb{R}$ (parameters of invocations, if any, are part of the invocation name). An *operation* is an invocation $i \in I$ paired with a return $r \in R$; we denote such an operation by $i \Rightarrow r$. We denote individual operations by $o$. The invocation, resp., the return, in an operation $o$ is denoted by $inv(o)$, resp., $ret(o)$.

The interaction between a client and an object is represented by a *history* $\langle po, hb \rangle$ over a set of operations $O$ which consists of

- a *program (order) po* which is a partial order on $O$, and
- a *happens-before (order) hb* which is a partial order on $O$.

The program order is enforced by the client, e.g., by invoking a set of operations within the same thread or process, while the happens-before order represents the order in which the operations finished, i.e., $(o_1, o_2) \in hb$ iff operation $o_1$ finished before $o_2$ started. We assume that the program order is included in the happens-before order.

*Example 1.* Let us consider a key-value map ADT containing operations of the form `put(key, value)` ⇒ `old`, which insert key-value pairs and return previously-mapped values for the given keys, `remove(key)` ⇒ `value`, which remove key mappings and return previously-mapped values, `contains(value)` ⇒ `true/false`, which test whether values are currently mapped, and `get(key)` ⇒ `value`, which return currently-mapped values for the given keys. Figure 1(a) pictures a history $h$ where edges denote the program order $po$ and happens-before $hb$. Such a history can be obtained by a client with three threads each making two invocations (the invocations within the same thread are aligned vertically).

The axiomatic specifications of concurrent objects we consider are based on the following abstract representation of executions: an *abstract execution* over operations $O$ is a tuple $\langle po, hb, lin, vis \rangle$ that consists of a history $\langle po, hb \rangle$ over $O$,

- a *linearization (order) lin*[2] which is a total order on $O$, and
- a *visibility (relation) vis* which is an acyclic relation on $O$.

---

[2] The linearization is also called *arbitration* in previous works, e.g., [9].

Intuitively, the visibility relation represents the inter-thread communication, how effects of operations are visible to other threads, while the linearization order models the "conflict resolution policy", how the effects of concurrent operations are ordered when they become visible to other threads.

We say that an operation $o_1$ such that $\langle o_1, o_2\rangle \in \mathit{vis}$ is *visible* to $o_2$, and that $o_2$ *sees* $o_1$. Also, the set of operations visible to $o_2$ is called the *visibility set* of $o_2$. The extensions of *inv* and *ret* to partial orders on $O$ are defined component-wise as usual.

*Example 2.* Figure 1(b) pictures an abstract execution containing the history in Fig. 1(a). The visibility relation is defined by the edges labeled *vis* together with their transitive closure. The linearization order is defined by the order in which operations are written (from top to bottom).

A consistency criterion for concurrent objects is defined by a set of axioms over the relations in an abstract execution. These axioms relate abstract executions to a sequential semantics of the operations, which is defined by a function $\mathit{Spec} : \mathbb{I}^* \times \mathbb{I} \to \mathbb{R}$ that determines the return value of an invocation given the sequence of invocations previously executed on the object[3].

*Example 3.* The sequential semantics of the key-value map ADT considered in Example 1 is defined as expected. For instance, the return value of $\texttt{put}(\texttt{key}, \texttt{value})$ after a sequence of invocations $\sigma$ is the value `null` if $\sigma$ contains no invocation $\texttt{put}(\texttt{key}, \ldots)$, or `old` if $\texttt{put}(\texttt{key}, \texttt{old})$ is the last invocation of the form $\texttt{put}(\texttt{key}, \ldots)$ in $\sigma$.

The *domain* $\mathrm{dom}(R)$ of a relation $R$ is the set of elements $x$ such that $\langle x, y\rangle \in R$ for some $y$; the *codomain* $\mathrm{codom}(R)$ is the set of elements $y$ such that $\langle x, y\rangle \in R$ for some $x$. By an abuse of notation, if $x$ is an individual element, $x \in R$ denotes the fact that $x \in \mathrm{dom}(R) \cup \mathrm{codom}(R)$. The *(left) composition* $R_1 \circ R_2$ of two binary relations $R_1$ and $R_2$ is the set of pairs $\langle x, z\rangle$ such that $\langle x, y\rangle \in R_1$ and $\langle y, z\rangle \in R_2$ for some $y$. We denote the identity binary relation $\{\langle x, x\rangle : x \in X\}$ on a set $X$ by $[X]$, and we write $[x]$ to denote $[\{x\}]$.

*Return-value consistency* [9], a variant of eventual consistency without liveness guarantees, states that the return $r$ of every operation $i \Rightarrow r$ can be obtained from a sequential execution of $i$ that follows the invocations visible to $o$ (in the linearization order). This constraint will be formalized as an axiom called $\mathsf{Ret}$. The visibility relation can be chosen arbitrarily. Standard "session guarantees" can be described in the same framework by adding constraints on the visibility relation: for instance, *read my writes*, i.e., operations previously executed in the same thread remain visible, can be stated as $\mathsf{vis} \supseteq \mathsf{po}$ and *monotonic reads*, i.e., the set of visible operations to some thread grows monotonically over time, can

---

[3] Previous works have considered more general, concurrent semantics for operations. We restrict ourselves to sequential semantics in order to simplify the exposition. Our results extend easily to the general case.

$$\begin{aligned}\phi &::= \mathsf{Ret} \mid \mathit{ord}\\ \mathit{ord} &::= \mathit{qrel} \supseteq \mathit{rel}\\ \mathit{qrel} &::= \mathsf{lin} \mid \mathsf{vis}\\ \mathit{rel} &::= \mathit{qrel} \mid \mathsf{po} \mid \mathsf{hb} \mid \mathit{rel} \circ \mathit{rel}\end{aligned}$$

**Fig. 2.** The grammar of consistency axioms.

$$\langle po, hb, lin, vis\rangle \models \mathsf{Ret} \text{ iff}$$
$$\forall o.\, ret(o) = Spec(inv(ctxt(lin, vis, o)), inv(o))$$

$$\langle po, hb, lin, vis\rangle \models ord \text{ iff}$$
$$ord[po/\mathsf{po}][hb/\mathsf{hb}][lin/\mathsf{lin}][vis/\mathsf{vis}] \text{ is valid}$$

**Fig. 3.** Consistency axiom satisfaction for abstract executions. The satisfaction relation $\models$ is implicitly parameterized by a sequential semantics *Spec* which we consider fixed.

be stated as $\mathsf{vis} \supseteq \mathsf{vis} \circ \mathsf{po}$. Then, a version of causal consistency [7,9], called *causal convergence*, is defined by the following set of axioms:

$$\mathsf{vis} \supseteq \mathsf{vis} \circ \mathsf{vis} \quad \mathsf{vis} \supseteq \mathsf{po} \quad \mathsf{lin} \supseteq \mathsf{vis} \quad \mathsf{Ret}$$

which state that the visibility relation is transitive, it includes program order, and it is included in the linearization order. Finally, *linearizability* is defined by the set of axioms $\mathsf{lin} \supseteq \mathsf{hb}$, $\mathsf{vis} = \mathsf{lin}$, and $\mathsf{Ret}$.

To state our results in a general context that concerns multiple consistency criteria defined in the literature (including the ones mentioned above) and variations there of, we consider a language of *consistency axioms* $\phi$ defined by the grammar in Fig. 2. A *consistency model* $\Phi$ is a set $\{\phi_1, \phi_2, \ldots\}$ of consistency axioms.

In the following, we assume that every consistency model is stronger than return-value consistency, and also, that the linearization order is consistent with the visibility and happens-before relations. The assumptions concerning the linearization order correspond to the fact that for instance, concurrent operations are ordered using timestamps that correspond to real-time. Formally, we assume that every consistency model contains the axioms

$$\Phi_0 = \{\mathsf{Ret}, \mathsf{lin} \supseteq \mathsf{vis}, \mathsf{lin} \supseteq \mathsf{hb}\}.$$

Figure 3 defines the precise semantics of consistency axioms on abstract executions: the *context* of an operation $o$ according to a linearization *lin* and visibility *vis*, denoted $ctxt(lin, vis, o)$ is the restriction $([O_o] \circ lin \circ [O_o])$ of *lin* to the operations $O_o = \mathrm{dom}(vis \circ [o])$ visible to $o$. For instance, for the abstract execution in Fig. 1(b), $ctxt(lin, vis, \mathtt{contains}(0) \Rightarrow \mathtt{false})$ is the sequence of operations $\mathtt{put}(1,0) \Rightarrow \mathtt{null}; \mathtt{get}(1) \Rightarrow 0; \mathtt{put}(1,1) \Rightarrow 0$.

We extend this semantics to consistency models as $e \models \Phi$ iff $e \models \phi$ for all $\phi \in \Phi$ and to histories as:

$$\langle po, hb\rangle \models \Phi \;\text{ iff }\; \exists lin, vis.\ \langle po, hb, lin, vis\rangle \models \Phi$$

*Example 4.* The abstract execution in Fig. 1(b) satisfies causal convergence: the visibility relation is transitive, it includes program order, and it is consistent with the linearization order. Moreover, the axiom Ret is also satisfied.

For instance, the invocation $\texttt{contains}(0)$ returns exactly `false` when executed after $\texttt{put}(1,0); \texttt{get}(1); \texttt{put}(1,1)$. Similarly, it returns `true` when executed after $\texttt{put}(1,0); \texttt{get}(1); \texttt{put}(0,0)$.

## 3 Minimal Visibility Extensions

Checking whether a given history satisfies a consistency model is intractable in general. This essentially follows from the fact that checking linearizability is NP-hard in general [18]. While the main issue in checking linearizability is enumerating the exponentially many linearizations, checking weaker criteria like causal convergence requires also an enumeration of the exponentially many visibility relations (included in a given linearization). We prove in this section that it is enough to enumerate only *minimal* visibility relations (w.r.t. set inclusion), included in a given linearization, in order to conclude whether a given history and linearization satisfy a consistency model.

A *linearized history* $\sigma = \langle po, hb, lin \rangle$ consists of a history and a linearization $lin$ such that $lin \supseteq hb$. The extension of $\models$ to linearized histories is defined as:

$$\langle po, hb, lin \rangle \models \Phi \ \text{ iff } \ \exists vis.\ \langle po, hb, lin, vis \rangle \models \Phi$$

The $i$-th element of a sequence $s$ is denoted by $s[i]$ and the prefix of $s$ of length $i$ is denoted by $s_i$. The projection of a linearized history $\sigma = \langle po, hb, lin \rangle$ to a prefix $lin_i$ of $lin$ is denoted by $\sigma_i$. Formally, $O_i = \mathrm{dom}(lin_i) \cup \mathrm{codom}(lin_i)$ and $\sigma_i = \langle po \cap (O_i \times O_i), hb \cap (O_i \times O_i), lin_i \rangle$.

For a linearized history $\langle po, hb, lin \rangle$ and a consistency model $\Phi$, a visibility relation $vis_i$ on operations from a prefix $lin_i$ of $lin$ is called *$\Phi$-extensible* when there exists a visibility relation $vis \supseteq vis_i$ such that $\langle po, hb, lin, vis \rangle \models \Phi$. The relation $vis$ is called a *$\Phi$-extension of $vis_i$ up to $lin$*. By extrapolation, a $\Phi$-extension of $vis_i$ *up to* $lin_j$ is a visibility relation $vis_j$ such that $\langle \sigma_j, vis_j \rangle \models \Phi$, for any $i < j$. Such an extension is called *minimal* when for every other $\Phi$-extension $vis'_j$ of $vis_i$ up to $lin_j$, we have that $vis'_j \not\subseteq vis_j$.

*Example 5.* Consider again the abstract execution in Fig. 1(b). Ignoring the edges labeled by $vis$, it becomes a linearized history $\sigma$. The prefix $\sigma_2$ contains just the two operations $\texttt{put}(1,0) \Rightarrow \texttt{null}$ and $\texttt{get}(1) \Rightarrow 0$. For causal convergence, the visibility relation $vis_2 = \{\langle \texttt{put}(1,0) \Rightarrow \texttt{null}, \texttt{get}(1) \Rightarrow 0 \rangle\}$ on operations of $\sigma_2$ is extensible, as witnessed by the visibility relation defined for the rest of the operations in this execution. The visibility relation

$$\begin{aligned} vis_3 = \{ & \langle \texttt{put}(1,0) \Rightarrow \texttt{null}, \texttt{get}(1) \Rightarrow 0 \rangle, \langle \texttt{put}(1,0) \Rightarrow \texttt{null}, \texttt{put}(0,0) \Rightarrow \texttt{null} \rangle, \\ & \langle \texttt{get}(1) \Rightarrow 0, \texttt{put}(0,0) \Rightarrow \texttt{null} \rangle \} \end{aligned}$$

is an extension of $vis_2$ up to $lin_3$, and contains the operations in $\sigma_2$ together with $\texttt{put}(0,0) \Rightarrow \texttt{null}$. Note that this extension is *not* minimal. A minimal extension would be exactly equal to $vis_2$ since, intuitively, $\texttt{put}(0,0) \Rightarrow \texttt{null}$ is not required to observe operations on keys other than 0.

The next lemma shows that minimizing the visibility sets of operations in a linearization prefix, while preserving the truth of the axioms on that prefix, doesn't exclude visibility choices for future operations (occurring beyond that prefix). In more precise terms, the $\Phi$-extensibility status is not affected by choosing smaller visibility sets for operations in a linearization prefix. For instance, since the visibility $vis_3$ in Example 5 is extensible (for causal convergence), the smaller visibility relation in which $\mathtt{put}(0,0) \Rightarrow \mathtt{null}$ doesn't see any operation, is also extensible. This result relies on the specific form of the axioms, which ensure that smaller visibility sets impose fewer constraints on the visibility sets of future operations. For instance, the axiom $vis \supseteq vis \circ vis$ enforces that $vis$ contains $\{\langle o, o_2\rangle : \langle o, o_1\rangle \in vis\}$ whenever a pair $\langle o_1, o_2\rangle$ is added to $vis$. Minimizing the visibility set of $o_1$ will minimize the set of operations that *must* be seen by $o_2$, thus making the choice of the operations visible to $o_2$ more liberal.

**Lemma 1.** *For every linearized history $\sigma$ and consistency model $\Phi$, if*

$$\langle \sigma_i, vis_i\rangle \models \Phi, \qquad vis_i \text{ is } \Phi\text{-extensible}, \qquad \langle \sigma_i, vis'_i\rangle \models \Phi, \qquad \text{and } vis'_i \subseteq vis_i,$$

*then $vis'_i$ is $\Phi$-extensible.*

*Proof (Sketch).* We show that the $\Phi$-extension $vis$ of $vis_i$ up to $lin$ can be transformed to a $\Phi$-extension of $vis'_i$ up to $lin$ by simply removing the pairs of operations in $vis_i \setminus vis'_i$. Let $vis'$ be this visibility relation and $\Phi$ a consistency model. We prove that $\langle po, hb, lin, vis'\rangle \models \Phi$ by considering the different types of axioms defined in Fig. 2.

Suppose that $\Phi$ contains an axiom of the form $\mathsf{vis} \supseteq rel$ (according to the notations in Fig. 2). We have that $vis'_i \supseteq (rel[po/\mathsf{po}][hb/\mathsf{hb}][lin/\mathsf{lin}][vis'/\mathsf{vis}]) \circ [O_i]$ by the hypothesis (from $(\sigma_i, vis'_i) \models \Phi$). Then, $vis'_i \subseteq vis_i$ implies that

$$\begin{aligned}&(rel[po/\mathsf{po}][hb/\mathsf{hb}][lin/\mathsf{lin}][vis/\mathsf{vis}]) \circ [O \setminus O_i]\\ &\supseteq (rel[po/\mathsf{po}][hb/\mathsf{hb}][lin/\mathsf{lin}][vis'/\mathsf{vis}]) \circ [O \setminus O_i]\end{aligned}$$

which together with $vis' \circ [O \setminus O_i] = vis \circ [O \setminus O_i]$ (the visibility relations $vis$ and $vis'$ are the same for operations which are not included in the prefix $lin_i$) implies that

$$vis' \circ [O \setminus O_i] \supseteq (rel[po/\mathsf{po}][hb/\mathsf{hb}][lin/\mathsf{lin}][vis'/\mathsf{vis}]) \circ [O \setminus O_i].$$

Therefore, $\langle po, hb, lin, vis'\rangle \models \mathsf{vis} \supseteq rel$.

The axiom $\mathsf{Ret}$ relates the return value of each operation $o$ in $\sigma$ to the set of operations visible to $o$. This relation is insensitive to the set of operations seen by an operation before $o$ in the linearization order. Therefore, $\langle po, hb, lin, vis'\rangle \models \mathsf{Ret}$ is an immediate consequence of $(\sigma_i, vis'_i) \models \mathsf{Ret}$ and the fact that $vis$ and $vis'$ are the same for operations which are not included in the prefix $lin_i$.

The axioms of the form $\mathsf{lin} \supseteq rel$ (according to the notations in Fig. 2) are straightforward implications of $\mathsf{lin} \supseteq \mathsf{hb}$ and $\mathsf{lin} \supseteq \mathsf{vis}$, which are assumed to be included in any consistency model. They hold for any linearized history. $\square$

The main result of this section shows that a visibility enumeration strategy that considers operations in the linearization order and computes minimal extensions iteratively, possibly backtracking to another choice of minimal extension if necessary, is complete in general (it finds a visibility relation satisfying the consistency axioms $\Phi$ iff the input linearized history satisfies $\Phi$). Backtracking is necessary since in general, there may exist multiple minimal extensions and all of them should be explored. For a given linearized history $\sigma$ and visibility relation *vis* on operations of $\sigma$, $vis_i = vis \circ [O_i]$ denotes the restriction of *vis* to operations from the prefix $lin_i$.

**Theorem 1.** *For every linearized history $\sigma$ and consistency model $\Phi$, $\sigma \models \Phi$ iff there exists a visibility relation vis such that*

$$\textit{for every } i,\ vis_{i+1} \textit{ is a minimal } \Phi\textit{-extension of } vis_i \textit{ up to } lin_{i+1}.$$

*Proof.* (Sketch) Let $\sigma$ be a linearized history such that $\sigma \models \Phi$. Therefore, there exists a visibility relation *vis* such that $\langle \sigma, vis \rangle \models \Phi$. We prove by induction that there exists a visibility relation $vis'$ satisfying the claim of the theorem. Assume that there exists a $\Phi$-extensible visibility relation $vis^j$ on operations in $lin_j$ which satisfies the claim of the theorem for every $i < j$ (we take $vis^0 = vis$). Let $vis^{j+1}$ be a minimal visibility relation on operations in $lin_{j+1}$ such that $vis^{j+1} \circ [O_j] = vis^j \circ [O_j]$ and $(\sigma_{j+1}, vis^{j+1}) \models \Phi$ (such a set exists because $vis^j$ is $\Phi$-extensible). By Lemma 1, $vis^{j+1}$ is $\Phi$-extensible. Also, $vis^{j+1}$ satisfies the claim of the theorem for every $i < j + 1$. The reverse direction is trivial. □

*Example 6.* In the context of the abstract execution in Fig. 1(b), the visibility relation defined by removing the *vis* edge ending in `put`$(0, 0) \Rightarrow$ `null`, and adding the transitive closure, satisfies the requirements in Theorem 1.

## 4 Efficient Monitoring of Consistency Models

We describe an algorithm for checking whether a given history satisfies a consistency model, which combines linearization enumeration strategies proposed in [29,38] with the visibility enumeration strategy proposed in Sect. 3.

The algorithm is defined by the procedure `checkConsistency` listed in Fig. 4. This recursive procedure searches for extensions of the input linearization and visibility (initially, `checkConsistency` will be called with $lin = vis = \emptyset$) which witness that the input history $h$ satisfies $\Phi$. It assumes that the inputs *lin* and *vis* satisfy the axioms of the consistency model $\Phi$ when the input history is projected on the linearized operations (the operations in *lin*). This projection is denoted by $h_{lin}$. Formally, the precondition of this procedure is that $\langle h_{lin}, lin, vis \rangle \models \Phi$.

The extensions of *lin* and *vis* are built in successive steps. At each step, the linearization is extended according to the procedure `linExtensions` and the visibility according to the procedure `visExtensions`.

The abstract implementation of `linExtensions`, presented in Fig. 4, chooses a set of *non-linearized* operations $O$ which are *minimal* among non-linearized

```
proc checkConsistency($h, \Phi, lin, vis$) {
  if (isComplete($h, lin$)) then
    return true;
  forall $lin'$ of linExtensions($h, lin$) do
    forall $vis'$ of visExtensions($h, lin', vis$) do
      if checkConsistency($h, \Phi, lin', vis'$) then
        return true;
  return false;
}
```

```
proc linExtensions($h, lin$) {
  let $O$ = minimals($h, lin$);
  forall $O'$ of subsets($O$)
    forall $seq$ of linearizations($O'$)
      let $lin'$ = append($lin, seq$);
      yield $lin'$;
}
```

```
proc visExtensions($h, lin, vis$){
  forall $vis'$ a minimal $\Phi$-extension
                  of $vis$ up to $lin$
    yield $vis'$;
}
```

**Fig. 4.** Checking consistency of a history. The procedures `linExtensions`, resp., `visExtensions` return the set of linearizations, resp., visibilities, produced by the instruction `yield`.

operations w.r.t. happens-before, i.e., returned by `minimals`($h$, $lin$), and appends any linearization of the operations in $O$ to the input linearization $lin$. Formally, $O \subseteq \{o : o \notin lin \text{ and } \forall o'.\ o' \notin lin \Rightarrow \neg o' \prec o\}$, where $\prec$ denotes the happens-before relation. The fact that the operations in $O$ are minimal among non-linearized operations ensures that the returned linearizations are consistent with the happens-before order.

Two linearization enumeration strategies proposed in the literature can be seen as instances of `linExtensions`. The strategy in [38] corresponds to the case where $O$ contains exactly one minimal operation. For instance, for the history in Fig. 1(a), this strategy will start by picking a minimal element in the happens-before relation, say `put`$(1, 0) \Rightarrow$ `null`, then, a minimal operation among the rest, say `get`$(1) \Rightarrow 0$, and so on.

The strategy proposed in [29] is slightly more involved (and according to experimental results, more efficient), but it relies on a presentation of histories $h$ as sequences of call and return actions (an operation spanning the time interval between its call and return action). The happens-before order is extracted as usual: an operation $o_1$ happens before an operation $o_2$ if its return occurs before the call of $o_2$. This strategy defines $O$ as the first non-linearized operation $o$ that returned in $h$ together with a set of non-linearized operations $O'$ that are concurrent with $o$ (i.e., are not ordered after $o$ in the happens-before order). The operation $o$ is linearized last in the returned extensions. For instance, consider the history $h$ in Fig. 5 represented as a sequence of call/return actions (small boxes at the begin, resp., end, of an interval denote call actions, resp., return actions). The first linearization extension (when $lin = \emptyset$) includes `put`$(1, 0) \Rightarrow$ `null` (the first operation to return) after some sequence of operations concurrent with it, for

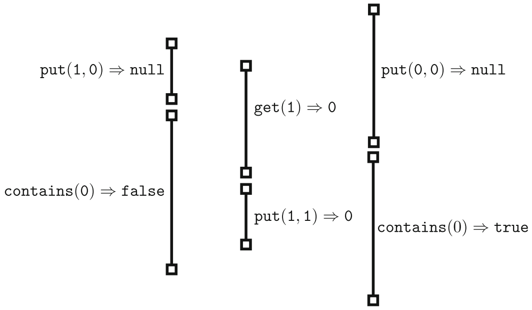


**Fig. 5.** The history $h$ in Fig. 1 presented as a sequence of call/return actions.

instance the empty sequence. Next, the current linearization $\texttt{put}(1,0) \Rightarrow \texttt{null}$ can be extended by adding $\texttt{put}(0,0) \Rightarrow \texttt{null}$ (the first operation to return, if we exclude $\texttt{put}(1,0) \Rightarrow \texttt{null}$ which is already linearized) and possibly $\texttt{get}(1) \Rightarrow 0$ before it. Suppose that we choose $\texttt{put}(1,0) \Rightarrow \texttt{null}; \texttt{get}(1) \Rightarrow 0; \texttt{put}(0,0) \Rightarrow \texttt{null}$. Then, the extension will include $\texttt{put}(1,1) \Rightarrow 0$ and possibly $\texttt{contains}(0) \Rightarrow \texttt{true}$ or $\texttt{contains}(0) \Rightarrow \texttt{false}$, and so on. Compared to the previous strategy, an extension step can add multiple operations.

The extensions of the visibility relation (returned by `visExtensions`) are minimal $\Phi$-extensions of *vis* up to the input linearization. They can be constructed iteratively by considering the newly linearized operations one by one and each time compute a minimal extension of the visibility. For instance, the linearization construction explained in the previous paragraph can be expanded with a visibility enumeration as follows:

- $lin = \texttt{put}(1,0) \Rightarrow \texttt{null}$: the minimal visibility is $vis_1 = \emptyset$,
- $lin = \texttt{put}(1,0) \Rightarrow \texttt{null}; \texttt{get}(1) \Rightarrow 0; \texttt{put}(0,0) \Rightarrow \texttt{null}$: the minimal visibility is $vis_2 = \{\langle \texttt{put}(1,0) \Rightarrow \texttt{null}, \texttt{get}(1) \Rightarrow 0\rangle\}$, and so on.

The procedure `checkConsistency` backtracks to a different extension when the current one cannot be completed to include all the operations in the input history (checked by the recursive call). The correctness of the algorithm is stated in the following theorem.

**Theorem 2.** $\texttt{checkConsistency}(h, \Phi, \emptyset, \emptyset)$ *returns true iff* $h \models \Phi$.

## 5 Empirical Results

While our minimal-visibility consistency checking algorithm is applicable to a wide class of distributed and multicore shared object implementations, here we demonstrate its efficacy on histories recorded from executions of Java Development Kit (JDK) Standard Edition concurrent data structures. Recent work demonstrates that JDK concurrent data structures regularly admit non-atomic behaviors, often by design [14]; these weakly-consistent behaviors span many methods of the `java.util.concurrent` package, including the ConcurrentHashMap, ConcurrentSkipListMap, ConcurrentSkipListSet, ConcurrentLinkedQueue, and the ConcurrentLinkedDeque, for instance, including the contains method described in Example 3.

We extracted 4,000 randomly-sampled histories from approximately 8,000 observed over approximately 1,000,000 executions in stress testing 20 randomly-generated client programs of the ConcurrentSkipListMap with up to 15 invocations across up to 3 threads. In each program, the given number of threads invokes its share of randomly-generated methods with randomly-generated values. We consider random generation superior to collecting programs *in the wild*, since found client programs can mask inconsistencies by restricting method argument values, or by being agnostic to inconsistent return values. Furthermore, automated generation gives us the ability to evaluate our algorithm on unbiased sample sets, and avoid any technical problems in the collection of programs; it also allows us to test method combinations which might not appear in publicly-available examples.

We subject each client program to 1 s of stress testing[4] to record histories. The return value of each invocation is stored in a different thread-local variable which is read at the end of the execution. Recording the happens-before order between invocations without affecting implementation behavior significantly (e.g., without influencing the memory orderings between shared-memory accesses) is challenging. For instance, we found the use of high-precision timers to be unsuitable, since the response-time of `System.nanoTime` calls is much higher than calls to the implementations under test; invoking such timers between each invocation of implementation methods would prevent implementation methods from overlapping in time, and thus hide any possible inconsistent behaviors. Similarly, the use of atomic operations and volatile variables would impose additional synchronization constraints and prevent many weak-memory reorderings.

Essentially, our solution is to introduce a shared variable per thread storing its program counter – in our context, the program counter stores the number of call and return events thus far executed. A thread's program counter is read by every other thread before and after each invocation. Figure 6 demonstrates a simplified version[5] of our encoding for a program with two threads each invoking two methods. The program counter variables `pc0` and `pc1` are not declared volatile, which, in principle, provides stronger guarantees concerning the derived happens-before relation; such declarations would interfere with implementation weak-memory effects. The program counter values read by each thread allows

---

[4] For stress testing we leverage OpenJDK's JCStress tool: http://openjdk.java.net/projects/code-tools/jcstress/.

[5] In our actual implementation, each program-counter access is encapsulated within a method call in order to avoid compiler reordering between the reads of other threads' counters and the increment of one's own. While the Java memory model does not guarantee that such encapsulation will prevent reordering, we found this solution to be adequate on Oracle's Java SE runtime version 9. Our actual implementation also wraps invocations in try-catch blocks to deal with exceptions.

```
int pc0 = 0, pc1 = 0;
ConcurrentHashMap obj = new ConcurrentHashMap();


void thread0() {                                void thread1() {
  Object r0, r1;                                  Object r0, r1;
  int pcs[][] = new int[4][1];                    int pcs[][] = new int[4][1];
  int n = 0;                                      int n = 0;

  // first invocation                             // first invocation
  pcs[n][0] = pc1; n++; pc0++;                    pcs[n][0] = pc0; n++; pc1++;
  r0 = obj.elements();                            r0 = obj.remove(1);
  pcs[n][0] = pc1; n++; pc0++;                    pcs[n][0] = pc0; n++; pc1++;

  // second invocation                            // second invocation
  pcs[n][0] = pc1; n++; pc0++;                    pcs[n][0] = pc0; n++; pc1++;
  r1 = obj.put(1,0);                              r1 = obj.put(0,1);
  pcs[n][0] = pc1; n++; pc0++;                    pcs[n][0] = pc0; n++; pc1++;

  // store the values of r0, r1, pcs              // store the values of r0, r1, pcs
  ...                                             ...
}                                               }
```

**Fig. 6.** Our encoding for recording ConcurrentHashMap histories. Each thread's program counter is read before and after other threads' invocations, and incremented subsequent to each such read. The two-dimensional `pcs`$[n][m]$ array stores $n$ program counter values for $m$ neighboring threads.

us to extract a happens-before order between invocations which is *sound* in the sense that the actual happens-before may order more operations, but not fewer – assuming that shared-memory accesses satisfy at least the total-store order (TSO) semantics in which writes are guaranteed to be performed according to program order. For instance, when $\texttt{pcs}[0][0] > 2$ in the second thread (`thread1`), the first invocation in the other thread (`thread0`) happens-before the first invocation in this thread. Otherwise, if $\texttt{pcs}[0][0] < 2$, then the two invocations are overlapping in time. The latter may not be true in the real happens-before due to the delay in incrementing and reading the program counter variables. Although some loss of precision is possible, we are unaware of other methods for tracking happens-before which avoid significant interference with the implementation under test.

Based on the encoding described above, we generate histories as sequences of call and return actions which serve as input to our consistency checking algorithms. For simplicity, we have considered just two consistency models, linearizability and a weak consistency model defined by $\{\mathsf{Ret}, \mathsf{lin} \supseteq \mathsf{vis}, \mathsf{lin} \supseteq \mathsf{hb}, \mathsf{vis} \supseteq \mathsf{hb}\}$ – see Sect. 2. We consider linearizability in order to measure the overhead of checking weak consistency due to visibility enumeration; the second model is simply the easiest weak-consistency model to support with our implementation; the choice among possible weak-consistency models appears fairly arbitrary, since the enumeration of visibility relations is common to all.

We consider several measurements, the results of which are listed in Figs. 7 and 8; all times are measured in milliseconds on logarithmic scale on a 2.7 GHz Intel Core i5 MacBook Pro with Oracle–s Java SE runtime version 9; and

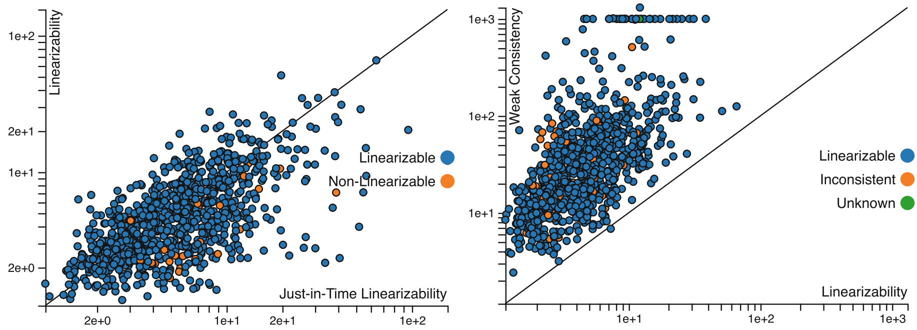


**Fig. 7.** Empirical comparison of (left) standard linearizability checking versus just-in-time linearizability checking on concurrent traces of Java data structures; and (right) weak-consistency checking versus standard linearizability checking. Each point reflects the time in milliseconds for checking a given trace.

timeouts are set to 1000 ms. We note that while accurate and *recording* of operation timings within an execution without interference is challenging, timing the *validation* of each recorded history, which we report here, is accomplished accurately, without interference, by computing the clock difference just before and after validation.

Our first measurements establish the baseline linearizability and weak-consistency checking algorithms. On the left side of Fig. 7 we consider the time required to check linearizability for each history by our own implementations of Wing and Gong's standard enumerative approach [38], along with Lowe's "just-in-time linearizability" algorithm [29] – see Sect. 4. We resolve the non-determinism in these algorithms (e.g., in choosing which pending operation to attempt linearizing first) arbitrarily (e.g., first called), finding no clear winner: each algorithm performs better on some histories. Since these subtleties are outside the scope of our work, we avoid further investigation and choose Wing and Gong's algorithm as our baseline linearizability-checking algorithm.

Our second measurement exposes the overhead of enumerating visibility relations for checking weak consistency. On the right side of Fig. 7 we consider the time required to check weak consistency of a given history versus the time required to check its linearizability.[6] We observe an overhead of approximately 10× due to visibility enumeration and validation. Our naïve implementation enumerates candidate visibilities in size-decreasing order since we expect visibility-loss to be the exception rather than the rule; for instance, atomic operations observe all linearized-before operations. We omit the analogous comparison between weak-consistency checking and just-in-time linearizability checking to avoid redundancy, since the just-in-time optimization is a seemingly-insignificant factor in our experiments: the results are nearly identical.

---

[6] Due to a benign error in the decoding of results of stress testing, we observe one single point on which the two algorithms conflict – labeled by "Unknown.".

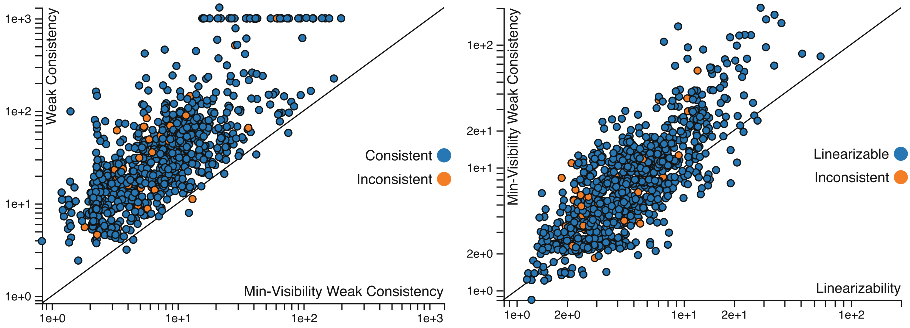


**Fig. 8.** Empirical comparison of (left) standard weak-consistency checking versus minimal-visibility weak-consistency checking on concurrent traces of Java data structures; and (right) the latter versus standard linearizability checking. Each point reflects the time in milliseconds for checking a given trace.

Our third measurement demonstrates the impact of our minimal-visibility consistency checking optimization. On the left side of Fig. 8 we consider the time required to check weak consistency without and with our optimization. The difference is dramatic, with our optimized algorithm consistently outperforming, sometimes up to multiple orders of magnitude: the leftmost 1000 ms timeout of the naïve algorithm is matched by a roughly 18 ms positive identification. Finally, our fourth measurement, on the right side of Fig. 8, demonstrates that the overhead of our minimal-visibility checking algorithm over linearizability checking is quite modest: we observe roughly a 2× overhead, compared with the observed 10× overhead without optimization.

While our experiments clearly demonstrate the efficacy of our minimal-visibility consistency checking algorithm, we will continue to evaluate this optimization across a wide range of concurrent objects, consistency models, and client programs, e.g., including many more concurrent threads. While we do expect the performance of linearizability- and weak-consistency checking to vary with thread count, we expect the performance gains of minimal-visibility consistency checking to continue to hold.

## 6 Related Work

Herlihy and Wing [22] described linearizability, which is the standard consistency criterion for shared-memory concurrent objects. Motivated by replication-based distributed systems, Burckhardt et al. [9,11] describe a more general axiomatic framework for specifying weaker consistencies like eventual consistency [36] and causal consistency [2]. Our weak consistency checking algorithm applies to consistency models described in this framework.

While several static techniques have been developed to prove linearizability [1,4,6,12,13,21,22,24,26,27,30–34,37,39], few have addressed dynamic techniques such as testing and runtime verification. The works in [29,38] describe monitors for checking linearizability that construct linearizations of a given history incrementally, in an online fashion. Line-Up [10] performs systematic concurrency testing via schedule enumeration, and offline linearizability checking via linearization enumeration. Our weak consistency checking algorithm combines these approaches with an efficient enumeration of visibility relations. The works in [15,16] propose a symbolic enumeration of linearizations based on a SAT solver. Although more efficient in practice, this approach applies only to certain ADTs. In this work, we propose a generic approach that assumes no constraints on the sequential semantics of the concurrent objects.

Bouajjani et al. [7] consider the problem of verifying causal consistency. They propose an algorithm for checking whether a given execution satisfies causal consistency, but only for the key-value map ADT with simple `put` and `get` operations. Our work proposes a generic algorithm that can deal with various weak consistency criteria and ADTs.

From the complexity standpoint, Gibbons and Korach [18] showed that monitoring even the single-value register type for linearizability is NP-hard. Alur et al. [3] showed that checking linearizability of all executions of a given implementation is in EXPSPACE when the number of concurrent operations is bounded, and then Hamza [20] established EXPSPACE-completeness. Bouajjani et al. [5] showed that the problem becomes undecidable once the number of concurrent operations is unbounded. Also, Bouajjani et al. [7,8] investigate various ADTs for which the problems of checking eventual and causal consistency are decidable.

## 7 Conclusion

We have developed the first completely-automatic algorithm for checking weak consistency of arbitrary concurrent object implementations which avoids the naïve enumeration of all possible visibility relations. While methodologies for constructing reliable yet weakly-consistent implementations are relatively immature, we believe that such implementations will continue to be important for the development of distributed and multicore software systems. Likewise, automation for testing and verifying such implementations is, and will increasingly be, important. Besides improving state-of-the-art verification algorithms, our results represent an important step for future research which may find other ways to exploit the soundness of considering only minimal visibilities, on which our optimized algorithm relies.

## References

1. Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 324–338. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_23
2. 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). https://doi.org/10.1007/BF01784241

3. Alur, R., McMillan, K.L., Peled, D.A.: Model-checking of correctness conditions for concurrent objects. Inf. Comput. **160**(1–2), 167–188 (2000). https://doi.org/10.1006/inco.1999.2847
4. Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 477–490. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_49
5. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Verifying concurrent programs against sequential specifications. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 290–309. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_17
6. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Tractable refinement checking for concurrent objects. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, 15–17 January 2015, Mumbai, India, pp. 651–662. ACM (2015). https://doi.org/10.1145/2676726.2677002
7. Bouajjani, A., Enea, C., Guerraoui, R., Hamza, J.: On verifying causal consistency. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, 18–20 January 2017, Paris, France, pp. 626–638. ACM (2017). http://dl.acm.org/citation.cfm?id=3009888
8. Bouajjani, A., Enea, C., Hamza, J.: Verifying eventual consistency of optimistic replication systems. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, 20–21 January 2014, San Diego, CA, USA, pp. 285–296. ACM (2014). https://doi.org/10.1145/2535838.2535877
9. Burckhardt, S.: Principles of eventual consistency. Found. Trends Program. Lang. **1**(1–2), 1–150 (2014). https://doi.org/10.1561/2500000011
10. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: Zorn, B.G., Aiken, A. (eds.) Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, 5–10 June 2010, Toronto, Ontario, Canada, pp. 330–340. ACM (2010). https://doi.org/10.1145/1806596.1806634
11. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, 20–21 January 2014, San Diego, CA, USA, pp. 271–284. ACM (2014). https://doi.org/10.1145/2535838.2535848
12. Dodds, M., Haas, A., Kirsch, C.M.: A scalable, correct time-stamped stack. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, 15–17 January 2015, Mumbai, India, pp. 233–246. ACM (2015). https://doi.org/10.1145/2676726.2676963
13. Drăgoi, C., Gupta, A., Henzinger, T.A.: Automatic linearizability proofs of concurrent objects with cooperating updates. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 174–190. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_11
14. Emmi, M., Enea, C.: Exposing non-atomic methods of concurrent objects. CoRR abs/1706.09305 (2017). http://arxiv.org/abs/1706.09305

15. Emmi, M., Enea, C.: Sound, complete, and tractable linearizability monitoring for concurrent collections. PACMPL **2**(POPL), 25:1–25:27 (2018). https://doi.org/10.1145/3158113
16. Emmi, M., Enea, C., Hamza, J.: Monitoring refinement via symbolic reasoning. In: Grove, D., Blackburn, S. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 15–17 June 2015, Portland, OR, USA, pp. 260–269. ACM (2015). https://doi.org/10.1145/2737924.2737983
17. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985). https://doi.org/10.1145/3149.214121
18. Gibbons, P.B., Korach, E.: Testing shared memories. SIAM J. Comput. **26**(4), 1208–1244 (1997). https://doi.org/10.1137/S0097539794279614
19. Gilbert, S., Lynch, N.A.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2), 51–59 (2002). https://doi.org/10.1145/564585.564601
20. Hamza, J.: On the complexity of linearizability. In: Bouajjani, A., Fauconnier, H. (eds.) NETYS 2015. LNCS, vol. 9466, pp. 308–321. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26850-7_21
21. Henzinger, T.A., Sezgin, A., Vafeiadis, V.: Aspect-oriented linearizability proofs. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013. LNCS, vol. 8052, pp. 242–256. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40184-8_18
22. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990). https://doi.org/10.1145/78969.78972
23. Kawell Jr., L., Beckhardt, S., Halvorsen, T., Ozzie, R., Greif, I.: Replicated document management in a group communication system. In: Proceedings of the 1988 ACM Conference on Computer-Supported Cooperative Work, p. 395. CSCW 1988. ACM, New York (1988). https://doi.org/10.1145/62266.1024798
24. Khyzha, A., Gotsman, A., Parkinson, M.: A generic logic for proving linearizability. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 426–443. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_26
25. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). https://doi.org/10.1145/359545.359563
26. Liang, H., Feng, X.: Modular verification of linearizability with non-fixed linearization points. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, 16–19 June 2013, Seattle, WA, USA, pp. 459–470. ACM (2013). https://doi.org/10.1145/2462156.2462189
27. 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
28. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In: Wobber, T., Druschel, P. (eds.) Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, 23–26 October 2011, Cascais, Portugal, pp. 401–416. ACM (2011). https://doi.org/10.1145/2043556.2043593
29. Lowe, G.: Testing for linearizability. Concurr. Comput.: Pract. Exp. **29**(4) (2017). https://doi.org/10.1002/cpe.3928

30. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: Richa, A.W., Guerraoui, R. (eds.) Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, 25–28 July 2010, Zurich, Switzerland, pp. 85–94. ACM (2010). https://doi.org/10.1145/1835698.1835722
31. Schellhorn, G., Wehrheim, H., Derrick, J.: How to prove algorithms linearisable. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 243–259. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_21
32. Sergey, I., Nanevski, A., Banerjee, A.: Mechanized verification of fine-grained concurrent programs. In: Grove, D., Blackburn, S. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 15–17 June 2015, Portland, OR, USA, pp. 77–87. ACM (2015). https://doi.org/10.1145/2737924.2737964
33. Sergey, I., Nanevski, A., Banerjee, A.: Specifying and verifying concurrent algorithms with histories and subjectivity. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 333–358. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_14
34. Shacham, O., Bronson, N.G., Aiken, A., Sagiv, M., Vechev, M.T., Yahav, E.: Testing atomicity of composed concurrent operations. In: Lopes, C.V., Fisher, K. (eds.) Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, 22–27 October 2011, Portland, OR, USA, pp. 51–64. ACM (2011). https://doi.org/10.1145/2048066.2048073
35. Terry, D.B., Demers, A.J., Petersen, K., Spreitzer, M.J., Theimer, M.M., Welch, B.B.: Session guarantees for weakly consistent replicated data. In: Proceedings of the Third International Conference on on Parallel and Distributed Information Systems, PDIS 1994, pp. 140–150. IEEE Computer Society Press, Los Alamitos (1994). http://dl.acm.org/citation.cfm?id=381992.383631
36. Terry, D.B., Theimer, M., Petersen, K., Demers, A.J., Spreitzer, M., Hauser, C.: Managing update conflicts in bayou, a weakly connected replicated storage system. In: Jones, M.B. (ed.) Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, 3–6 December 1995, Copper Mountain Resort, Colorado, USA, pp. 172–183. ACM (1995). https://doi.org/10.1145/224056.224070
37. 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
38. Wing, J.M., Gong, C.: Testing and verifying concurrent objects. J. Parallel Distrib. Comput. **17**(1–2), 164–182 (1993). https://doi.org/10.1006/jpdc.1993.1015
39. Zhang, S.J.: Scalable automatic linearizability checking. In: Taylor, R.N., Gall, H.C., Medvidovic, N. (eds.) Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, 21–28 May 2011, Waikiki, Honolulu, HI, USA, pp. 1185–1187. ACM (2011). https://doi.org/10.1145/1985793.1986037