Sound, Complete, and Tractable Linearizability Monitoring for Concurrent Collections

MICHAEL EMMI, SRI International, USA
CONSTANTIN ENEA, IRIF, Univ. Paris Diderot & CNRS, France

While many program properties like the validity of assertions and in-bounds array accesses admit nearly-trivial monitoring algorithms, the standard correctness criterion for concurrent data structures does not. Given an implementation of an arbitrary abstract data type, checking whether the operations invoked in one single concurrent execution are *linearizable*, i.e., indistinguishable from an execution where the same operations are invoked atomically, requires exponential time in the number of operations.

In this work we identify a class of *collection* abstract data types which admit polynomial-time linearizability monitors. Collections capture the majority of concurrent data structures available in practice, including stacks, queues, sets, and maps. Although monitoring executions of arbitrary abstract data types requires enumerating exponentially-many possible linearizations, collections enjoy combinatorial properties which avoid the enumeration. We leverage these properties to reduce linearizability to Horn satisfiability. As far as we know, ours is the first sound, complete, and tractable algorithm for monitoring linearizability for types beyond single-value registers.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; • **Theory of computation** → *Logic and verification*; • **Computing methodologies** → Concurrent algorithms;

Additional Key Words and Phrases: Linearizability; Runtime Verification; Concurrent Objects

ACM Reference Format:
Michael Emmi and Constantin Enea. 2018. Sound, Complete, and Tractable Linearizability Monitoring for Concurrent Collections. *Proc. ACM Program. Lang.* 2, POPL, Article 25 (January 2018), 28 pages. https://doi.org/10.1145/3158113

1 INTRODUCTION

Efficient multithreaded programs typically rely on optimized implementations of common abstract data types (ADTs) like stacks, queues, sets, and maps [Liskov and Zilles 1974], whose operations execute in parallel across processor cores to maximize efficiency [Moir and Shavit 2004]. Programming these "concurrent data structures" is tricky though. Synchronization between operations must be minimized to reduce response time and increase throughput [Herlihy and Shavit 2008; Moir and Shavit 2004]. Yet this minimal amount of synchronization must also be adequate to ensure that operations behave as if they were executed atomically, one after the other, so that client programs can rely on their (sequential) ADT specification; this de-facto correctness criterion is known as *linearizability* [Herlihy and Wing 1990]. These opposing requirements, along with the general challenge in reasoning about thread interleavings, make concurrent data structures a ripe source of insidious programming errors [Michael 2004].

Authors' addresses: Michael Emmi, SRI International, USA, michael.emmi@sri.com; Constantin Enea, IRIF, Univ. Paris Diderot & CNRS, France, cenea@irif.fr.

2475-1421/2018/1-ART25
https://doi.org/10.1145/3158113

Program properties like linearizability that are difficult to determine statically are typically substantiated by dynamic techniques like testing and runtime verification. Besides increasing confidence in the validity of such properties, property monitoring enables the eager diagnosis of anomalies which may otherwise only exhibit far-removed manifestations. However, while typical safety properties like the validity of assertions and in-bound array accesses admit nearly-trivial monitors which require only linear time in the length of program executions [Havelund and Rosu 2004], monitoring linearizability of an execution against an arbitrary ADT specification requires exponential time [Gibbons and Korach 1997]. This lower-bound limits precise monitoring to very short executions with few operations [Burckhardt et al. 2010], ultimately rendering testing and runtime verification ineffective for concurrent data structures, and limiting practical utilization to a small number of trusted implementations.

In this work we identify a class of ADTs, called *collection types*, which admit polynomial-time linearizability monitors. Intuitively, collection types capture the typical collection-based data structures, including the stacks, queues, sets, and maps provided by libraries like `java.util.concurrent`. Technically, we characterize collection types as containers of values whose operations add, remove, and use values. In addition, collection types satisfy four semantic properties: *value invariance* excludes types whose operations have effects beyond adding and removing values; *locality* excludes types whose operations add or use values beyond those appearing as arguments or returns; *parametricity* excludes types whose operations use values un-opaquely; *reducibility* excludes types which are not fully characterized by small representative behaviors. While in practice concurrent data structures often provide auxiliary operations which are not value invariant, local, or reducible, such methods often eschew linearizability in the name of efficiency. For example, while the size method of a collection ADT is typically neither local nor reducible, it is also not typically linearizable because the cost of synchronization required for atomically tallying, or maintaining a tally of, all elements apparently outweighs the need for a precise count. Recent work demonstrates that the majority of these auxiliary methods are not linearizable [Emmi and Enea 2017], and thus our focus on ensuring linearizability for value-invariant, local, parametric, and reducible methods of a collection ADT may not amount to a significant practical restriction.

The challenge we face in demonstrating tractability of the linearizability problem for collection types is avoiding an exponential enumeration. Since the operations of a concurrent data structure are permitted to execute in parallel, the happens-before order among the operations of any given execution is generally only a partial linearization, i.e., a partial order. This partial linearization is consistent with a given ADT when it can be extended to a total linearization whose resulting sequence of operations is admitted. However, there are generally an exponential number of total extensions to a given partial linearization, and for an arbitrary ADT, enumerating each extension is asymptotically optimal [Gibbons and Korach 1997].

Our result establishes that the exponential enumeration of total linearizations is avoidable for collection types. Our proof is essentially by reduction to a logical satisfiability problem in which models represent linearizations, and formulas encode linearization constraints, including an axiomatization of total orders to ensure that the resulting linearization is total. The input to this satisfiability problem is an *operation-order specification* (OOS) which amounts to an encoding of the constraints of a given ADT as a conjunction of Horn clauses, i.e., of disjunctions with at most one positive literal. Unlike formulas with arbitrary propositional structure, satisfiability of conjunctions of Horn clauses is tractable [Dowling and Gallier 1984]. Intuitively, Horn satisfiability avoids the exponential enumeration of possible satisfying assignments, corresponding to the exponential enumeration of total linearizations. However, the totality axioms require clauses like $x = y \vee x < y \vee y < x$ which are not Horn. The crux of our result lies in the *Hornification* of formulas which include total-order axiomatization: they can be rewritten to equisatisfiable Horn formulas.

The reduction to this tractable logical satisfiability problem — called OOS *linearizability* — is another challenge. Essentially, we must show that the semantic properties which define collection types guarantee the existence of sound operation-order specifications, so that OOS linearizability is equivalent to linearizability.

We fulfill this reduction by showing that the locality and parametricity properties ensure a nice combinatorial property: when arbitrarily-large operation sequences embed, in a certain precise sense, subsequences which are violations, i.e., are not admitted by the collection type, the larger sequences are themselves violations. Furthermore, the value-invariance and reducibility properties ensure that there are a finite number of minimal violations, which can thus be thought of as the atomic patterns which every violation embeds, and that these patterns are bounded in size. Collectively, these combinatorial properties ensure that collection types have sound operation-order specifications, with constraints corresponding to the exclusion of violation patterns.

To the best of our knowledge, our result yields the first tractable, sound, and complete algorithm for checking linearizability for collection ADTs, representing the majority of practically-available concurrent data structures. Our work is complimented by others which demonstrate effective implementations of inference [Emmi and Enea 2016] and runtime monitoring [Emmi et al. 2015] of operation-order specifications. In particular [Emmi et al. 2015] evaluated the performance of an implementation of the algorithm we prove sound, demonstrating orders of magnitude improvement in runtime verification. While these works conjectured their soundness (i.e., the guarantee that every linearizability violation is identified) based on empirical evidence, our result amounts to the first conclusive proof of their soundness.

Our contributions and outline are summarized as follows:

§3 identifies the class of collection types, demonstrates naturally-occurring instances like stacks, queues, sets, and maps, and states our main result on polynomial-time linearizability monitoring.
§4 reduces linearizability to a logical satisfiability problem requiring sound operation-order specifications.
§5 proves that linearizability against operation-order specifications is tractable, by reduction to Horn satisfiability.
§6 demonstrates key combinatorial properties of collection types enabling characterization via violation patterns.
§7 demonstrates that sound operation-order specifications are guaranteed to exist for collection types.

Section 2 recalls the linearizability problem and its intractability, Sections 8 and 9 conclude with a discussion of related work and concluding remarks.

2 LINEARIZABILITY

Efficient implementations of concurrent software components use fine-grain atomic operations to allow their methods to execute concurrently [Herlihy and Shavit 2008]. Reasoning about the correctness of such implementations thus requires specifying the expected behavior of concurrently-executed methods. Typical specification mechanisms for sequentially-executed methods describe the pre- and post-conditions of methods executed in isolation [Hoare 1969]. To reduce the meanings of concurrent behaviors to those of sequential behaviors, linearizability [Herlihy and Wing 1990] demands that some linearization of the only partially-ordered observed concurrent method invocations must correspond to a valid sequential behavior. In this section we formalize notions of behavior and specification which define the linarizability problem addressed in the remainder.

Accordingly, an *invocation label* $m(\vec{v})$ is a method name m along with a vector $\vec{v}$ of argument values. An *operation label* $\ell = m(\vec{v}) \Rightarrow v$ is an invocation label $m(\vec{v})$ along with a return value v. We write $\mathrm{args}(\ell)$ to denote the multiset of argument values $\biguplus_i v_i$, and $\mathrm{vals}(\ell)$ to denote the multiset of argument and return values $\mathrm{args}(\ell) \cup \{v\}$. An *interface* is a set of operation labels over a finite set of method names. In this work we consider only *fixed-arity* interfaces, i.e., the number of method arguments is fixed, where argument and return values have opaque infinite-domain polymorphic types, i.e., which can be instantiated with arbitrary datatypes including integers, strings, and object references.

Remark 2.1. While our simplified formalization allows only argument and return values from infinite domains, our results are easily extended to consider values from any finite domain as well, e.g., Boolean values, e.g., null values. There is no loss of generality in supposing method names include such finite-domain return values since we consider completed operations only. For instance, a set type would include operations void-type operations $\mathrm{containsTrue}(v)$ and $\mathrm{containsFalse}(v)$ in place of Boolean operations $\mathrm{contains}(v) \Rightarrow b$.

Definition 2.2. An *abstract data type (*ADT*)* is a prefix-closed set of operation-label sequences over a fixed-arity interface.

Example 2.3. The stack type is the set of sequences of $\mathrm{push}(v)$ and $\mathrm{pop} \Rightarrow v$ operations including

$$\mathrm{push}(1) \cdot \mathrm{push}(2) \cdot \mathrm{pop} \Rightarrow 2 \cdot \mathrm{pop} \Rightarrow 1 \text{ and}$$
$$\mathrm{push}(1) \cdot \mathrm{pop} \Rightarrow 1 \cdot \mathrm{push}(2) \cdot \mathrm{pop} \Rightarrow 2$$

in which the response of each pop operation matches the most-recent unmatched push operation's argument. Queues with $\mathrm{enqueue}(v)$ and $\mathrm{dequeue} \Rightarrow v$ operations are described similarly. Following Remark 2.1, the set type includes the operations $\mathrm{addTrue}(v)$, $\mathrm{removeTrue}(v)$, $\mathrm{containsTrue}(v)$, and their counterparts, $\mathrm{addFalse}(v)$, etc.

Example 2.4. The key-value map type includes $\mathrm{put}(k, v) \Rightarrow b$ operations, binding the key k to value v and returning a Boolean b indicating whether a binding of key k is overwritten, along with $\mathrm{remove}(k) \Rightarrow v$ operations, removing and returning a possibly-null (see Remark 2.1) value v bound to key k, and $\mathrm{get}(k) \Rightarrow v$ operations, returning the possibly-null value v bound to key k.

A *history* $\langle O, \prec \rangle$ is a function O mapping operations to operation labels, along with a happens-before partial-order $\prec$ over the operations $\mathrm{dom}(O)$. A *linearization* of a history h is a sequence s of operation labels such that:

- s and h have the same labels: $s_i = O(f(i))$, and
- s preserves the order of h: $i < j$ if $f(i) \prec f(j)$,

for some bijection $f : |\mathrm{dom}(O)| \to \mathrm{dom}(O)$ between indices and operations. In the worst case there are $O(k^n)$ linearizations of a history with n operations of which at most k are mutually unordered.

Definition 2.5. The *linearizability problem* asks whether a given ADT includes some linearization of a given history.

Example 2.6. The history with concurrent operations $\mathrm{push}(1)$ and $\mathrm{push}(2)$ preceding $\mathrm{pop} \Rightarrow 1$ is linearizable since the stack type admits the linearization $\mathrm{push}(2) \cdot \mathrm{push}(1) \cdot \mathrm{pop} \Rightarrow 1$. The history would not be linearizable were $\mathrm{push}(1)$ to precede $\mathrm{push}(2)$, since the stack type does not admit its only possible linearization, $\mathrm{push}(1) \cdot \mathrm{push}(2) \cdot \mathrm{pop} \Rightarrow 1$.

Adept readers may notice that our definition assumes all operations are completed. This is for technical convenience only. The linearizability problem remains NP-hard even when all operations are completed, and the forthcoming tractability results of our work hold for an arbitrary number of concurrent operations, so long as the number of non-read-only pending operations at the time of a linearizability check is bounded.

THEOREM 2.7. *The linearizability problem is* NP*-complete.*

PROOF. See Gibbons and Korach [1997]. □

While this work is devoted to the analysis of individual histories, one can also consider the analysis of ADT implementations. An implementation is essentially a transition system with call actions, corresponding to method invocations from client programs, and return actions, corresponding to method returns. Each execution corresponds to a single history in which each pair of operations is ordered if the first's return occurs before second's invocation. The set of all possible executions over all possible clients thus yields a set of histories associated with an implementation. Taking an abstract view, we say an *implementation* of an ADT is a set of histories, and that the implementation is *linearizable* if all of its histories are linearizable according to the ADT. We leverage this concept in Section 4 to form a notion of soundness of our approach to linearizability checking of individual histories.

3 COLLECTION TYPES

In this section we identify a class of *collection* abstract data types which admit polynomial-time linearizability monitors. We characterize this class with a semantic notion of the state of an arbitrary type as a container of values. This notion of state gives rise to other semantic notions, such as whether a given operation adds a given value to the state or uses a given value from the state. We build from these semantic notions the four properties that define collections: *value invariance*, *locality*, *parametricity*, *reducibility*. Finally, we conclude the section by stating our main result.

We begin by developing a notion of observation-based states. An *observation* from a sequence $s \in T$ of an abstract data type T is an operation-label sequence s' such that $s \cdot s' \in T$. Two sequences $s, s' \in T$ are *distinguishable* by an observation s'' when $s \cdot s'' \in T$ and $s' \cdot s'' \notin T$ or vice versa. Two sequences $s, s' \in T$ are *equivalent*, written $s \equiv s'$, when they are not distinguishable by any observation.

Example 3.1. Consider the following stack behaviors:

$$\begin{aligned}&\mathrm{push}(1) \cdot \mathrm{push}(2),\\ &\mathrm{push}(3) \cdot \mathrm{push}(2), \text{ and}\\ &\mathrm{push}(1) \cdot \mathrm{push}(2) \cdot \mathrm{push}(3) \cdot \mathrm{pop} \Rightarrow 3.\end{aligned}$$

While the first two sequences are distinguishable by the observation $\mathrm{pop} \Rightarrow 2 \cdot \mathrm{pop} \Rightarrow 1$, the first and third sequences are equivalent.

We represent states with canonical representatives of equivalence classes of observations. Given any total order $\ll$ on operation labels, e.g., lexicographic order over method names and argument and return values $m \cdot \vec{v} \cdot v$, we lift $\ll$ to operation-label sequences, ordering equal-length sequences by lexicographic order, and ordering unequal-length sequences by their length; i.e., $s \ll s'$ iff $|s| < |s'|$ or $|s| = |s'|$ and there exists i for which $s_i \ll s'_i$ and $s_j = s'_j$ for all $j < i$. The *state*, or *construction sequence*, $[\![s]\!]$ of an operation-label sequence s is the minimum equivalent operation-label sequence:

$$[\![s]\!] = \min_{\ll} \{s' : s \equiv s'\}.$$

$$\begin{aligned}
\text{readonly}(s \cdot \ell) &\Leftrightarrow [\![s \cdot \ell]\!] = [\![s]\!] \\
\text{adds}(s \cdot \ell, v) &\Leftrightarrow v \in (\text{contents}(s \cdot \ell) \setminus \text{contents}(s)) \\
\text{removes}(s \cdot \ell, v) &\Leftrightarrow v \in (\text{contents}(s) \setminus \text{contents}(s \cdot \ell)) \\
\text{uses}(s \cdot \ell, v) &\Leftrightarrow \text{drop}(s, \{v\}) \cdot \ell \notin T \\
\text{touches}(s \cdot \ell, v) &\Leftrightarrow \text{adds}(s \cdot \ell, v) \lor \text{uses}(s \cdot \ell, v)
\end{aligned}$$

Fig. 1. Operation predicate semantics.

Thus we represent the state after a given operation sequence as the minimal sequence which yields the same observations.

We associate each sequence with a multiset of values that its state contains: those which occur as arguments in the minimum equivalent sequence. We write args(s) to denote the multiset $\biguplus_i \text{args}(\ell_i)$ of argument values in the sequence $s = \ell_0\ell_1 \ldots$; the *contents* of a sequence s is the multiset contents(s) = args($[\![s]\!]$) of arguments to the construction sequence $[\![s]\!]$. Intuitively, the argument values of $[\![s]\!]$ define the values contained after executing s; the responses of subsequent method invocations would not include any removed value v, and thus the state, being a minimal sequence, would not contain operations adding v.

Example 3.2. The following stack behaviors

$$\begin{aligned}
&\text{push}(2), \\
&\text{push}(1) \cdot \text{pop} \Rightarrow 1 \cdot \text{push}(2), \text{ and} \\
&\text{push}(1) \cdot \text{pop} \Rightarrow 1 \cdot \text{push}(2) \cdot \text{push}(3) \cdot \text{pop} \Rightarrow 3
\end{aligned}$$

all have the same state, push(2).

We define the class of collection types as properties on operation sequences. An *operation predicate* is a predicate $P(s, \vec{v})$ on non-empty operation-label sequences $s = s' \cdot \ell$ and values $\vec{v}$ denoting a property of the operation labeled ℓ from the context of the operation-label sequence s'. The *occurrences* of an operation predicate P in operation-label sequence s over value vector $\vec{v}$ is the set

$$P^*(s, \vec{v}) = \{i : P(s_0 \cdots s_i, \vec{v})\}$$

of operation indices at which P holds.

Figure 1 lists the semantics of several operation predicates. We write drop(s, V) to denote the maximal subsequence s' of s such that vals(s') $\cap V = \emptyset$, where vals(s) denotes the multiset $\biguplus_i \text{vals}(\ell_i)$ of argument and return values in the sequence $s = \ell_0\ell_1 \ldots$. We treat unary operation predicates as adjectives, e.g., saying an operation is "read-only," and binary predicates as verbs, e.g., saying an operation "adds" a given value. An operation labeled ℓ following a sequence s is read-only when it leaves the construction sequence unchanged; the operation adds or removes a value v when the difference in contents before and after includes v; the operation uses v when its response is invalid were v not present; finally, the operation touches v when it either adds v or uses v.

Example 3.3. The push(v) operations of the stack type add v, while the pop $\Rightarrow v$ operations remove and use v. A peek $\Rightarrow v$ operation, returning the most-recently pushed value v, would be read-only and use v.

The first property of collection types excludes operations which have any affect besides adding or removing values. Technically, a type is *value invariant* when each operation which uses a value v either removes v or is read-only.

Example 3.4. The stack type with push, pop, and peek operations is value invariant. A stack type which supported an operation that increments the most-recently pushed (integer) value would not be value invariant. Similarly, the queue, set, and map types in Examples 2.3–2.4 are value invariant.

The second property of collection types excludes operations which touch values that do not appear as argument or return values. Technically, a *local interpretation* for an operation predicate P and abstract data type T is a predicate P' on individual operation labels such that for all sequences $s \cdot \ell \in T$ we have $P(s \cdot \ell, \vec{v})$ iff $P'(\ell, \vec{v})$. We say that a type is *local* when local interpretations exist for the readonly, adds, and uses predicates[1] of Figure 1, and its operations touch exactly the values appearing in its label, i.e., $v \in \text{vals}(\ell) \Leftrightarrow \text{touches}(\ell, v)$.

Example 3.5. The stack type with push, pop, and peek operations is local because push(v), pop $\Rightarrow v$, and peek $\Rightarrow v$ touch only value v. A stack type which supported a size $\Rightarrow n$ operation would not be local since its response depends on n values, none of which appear as arguments or returns. Similarly, the queue, set, and map types in Examples 2.3–2.4 are local.

The third property of collection types excludes those which do not treat values opaquely. Technically, a *(conflict-free) substitution* σ is a (injective) function from values to values. We lift σ to operation labels point-wise,

$$\sigma(m(u_1, \ldots, u_n) \Rightarrow v) = m(\sigma(u_1), \ldots, \sigma(u_n)) \Rightarrow \sigma(v),$$

and likewise to operation-label sequences,

$$\sigma(s) = \sigma(s_0) \cdot \sigma(s_1) \cdots,$$

and histories, via function composition $\circ$,

$$\sigma(\langle O, \prec \rangle) = \langle \sigma \circ O, \prec \rangle.$$

An abstract data type is *parametric* if it admits $\sigma(s)$ for all substitutions σ and admitted sequences s.

Example 3.6. The stack type is parametric since any renaming, e.g., $\{1 \mapsto 4, 2 \mapsto 3, 3 \mapsto 8\}$, applied to any admitted behavior, e.g., push(1) $\cdot$ push(2) $\cdot$ pop $\Rightarrow$ 2, yields another admitted behavior, e.g., push(4) $\cdot$ push(3) $\cdot$ pop $\Rightarrow$ 3. A stack type which supported an operation that replaces the two most-recently pushed (integer) values with their sum would not be parametric, e.g., since the renaming above applied to push(1) $\cdot$ push(2) $\cdot$ sum $\cdot$ pop $\Rightarrow$ 3 yields the behavior push(4) $\cdot$ push(3) $\cdot$ sum $\cdot$ pop $\Rightarrow$ 8, which is not admitted. Similarly, the queue and set types of Example 2.3 are parametric as well. In the case of sets, the Boolean return values are encoded in the name of the methods (see Remark 2.1), and they are therefore not changed by a substitution σ.

Remark 3.7. Our notion of parametricity covers set types with add, remove, and contains methods, which are problematic for other works [Abdulla et al. 2013; Henzinger et al. 2013] because their notion of "data independence" considers the sequence addTrue(v) $\cdot$ addFalse(v) to be ambiguous since v appears syntactically as the argument to both adds. Our semantic characterization avoids

[1] As will become clear in Section 7, local interpretations of readonly, adds, and uses are necessary to determine in isolation whether each concurrently-executing operation is read-only, adds a value, or uses a value; the touches predicate is derived from adds and uses; local interpretation of the removes predicate is unneeded. The removes predicate is important for value invariance, which is used to ensure that added values are only removed and not otherwise modified.

this problem; the sequence is unambiguous because the second add operation does not actually add v.

Example 3.8. Treating the key-value map type described in Example 2.4 as a collection requires extending our simplified formalization and considering key-value pairs as the unit of values contained in the collection. We then describe parametricity in terms of (conflict-free) key-value substitution pairs $\sigma = \langle \sigma_1, \sigma_2 \rangle$. We lift σ to operations by applying σ_1 or σ_2 depending on operation argument and return types; for instance, $\sigma(\mathrm{put}(k, v) \Rightarrow v') = \mathrm{put}(\sigma_1(k), \sigma_2(v)) \Rightarrow \sigma_2(v')$.

The fourth property of collection types excludes those which cannot be characterized by small representative behaviors. Technically, we write $\mathrm{keep}(s, V)$ to denote the maximal subsequence s' of s such that $\mathrm{vals}(s') \subseteq V$. Then we say that type T is *reducible* when there exists some bound $k \in \mathbb{N}$ such that $s \in T$ whenever $\mathrm{keep}(s, V) \in T$ for every subset $V \subseteq \mathrm{vals}(s)$ of size $|V| \le k$. Intuitively, the admittance of behaviors of reducible types is determined by considering the admittance of all sub-behaviors with at most k-values.

Example 3.9. The stack type is reducible with $n = 2$, since the admittance of any behavior, e.g.,

$$\mathrm{push}(1) \cdot \mathrm{push}(2) \cdot \mathrm{pop} \Rightarrow 2 \cdot \mathrm{push}(3) \cdot \mathrm{pop} \Rightarrow 3$$

is guaranteed when all maximal 2-value subsequences are admitted, e.g., when

$$\begin{aligned} &\mathrm{push}(1) \cdot \mathrm{push}(2) \cdot \mathrm{pop} \Rightarrow 2, \\ &\mathrm{push}(1) \cdot \mathrm{push}(3) \cdot \mathrm{pop} \Rightarrow 3, \text{ and} \\ &\mathrm{push}(2) \cdot \mathrm{pop} \Rightarrow 2 \cdot \mathrm{push}(3) \cdot \mathrm{pop} \Rightarrow 3 \end{aligned}$$

are admitted. Intuitively, types like stacks and queues are reducible with $n = 2$ because the correct response to each pop and dequeue operation is validated by considering each pair of values in isolation, i.e., to determine the least- or most-recent value added, respectively. A type whose removes are allowed to return any value except the first- or last-added value would be reducible with $n = 3$, yet not with $n = 2$. Similarly, the set and map types of Examples 2.3–2.4 are reducible with $n = 2$.

Remark 3.10. While reducibility allows focusing attention on small sub-behaviors, a given implementation may only exhibit violating sub-behaviors embedded within much large behaviors. For instance, one expects a queue implemented with a circular array of length 100 to exhibit a violation only when capacity exceeds 100. Reducibility does not imply violations are exhibited in small executions.

Finally the class of collections contains exactly the types which satisfy the aforementioned properties.

Definition 3.11. A *collection* is an abstract data type which is value invariant, local, parametric, and reducible.

Remark 3.12. The class of collection types captures the core methods of many concurrent data structures available in practice. Our analysis of the stack type being a collection is also applicable to queues, sets, and key-value maps types, as described by Examples 2.3–2.4. Although such concurrent data structures often provide other auxiliary operations which may not be value invariant, local, or reducible, such operations often compromise linearizability to achieve efficiency. For example, the size method of a collection ADT would not be local, intuitively since it uses all of the values contained in the collection, nor reducible, intuitively since a violation of size amounts to an incorrect return value, which is not generally preserved in the absence of surrounding operations.

However, the size method is also not typically linearizable because the cost of synchronization required for atomically tallying, or maintaining a tally of, all contained values apparently outweighs the utility of precise counting. Recent work demonstrates that the majority of these auxiliary methods are not linearizable [Emmi and Enea 2017], and so our focus on ensuring linearizability for collection-type methods is with little if any loss of generality. Furthermore, it is also possible that non-local and irreducible methods do not generally admit efficient linearizable implementations; similar conclusions have been made in the context of distributed systems, negatively correlating consistency, availability, and partition-tolerance [Brewer 2000], and in the context of operating systems, positively correlating the scalability of kernel operations to their commutativity [Clements et al. 2015]. Finally, it is worth noting that our resulting monitoring algorithm can be applied to any execution in which the only non-collection-type methods invoked are read-only; for instance, invocations of read-only size methods would be allowed, but invocations of mutating clear methods would not be.

The main result of our work is the following theorem stating that linearizability is tractable for collection types. Technically, our result requires the ability to differentiate between distinct instances of the same value. We say that an operation-label sequence s is *unambiguous* when at most one operation adds each value v, i.e., $|\text{adds}^*(s, v)| \leq 1$, and a history is *unambiguous* when its linearizations are. Note that extensions to parametricity like that of Example 3.8 necessitate a corresponding extension to unambiguity, e.g., that at most one operation adds each *bound value* v, i.e., $\sum_k |\text{adds}^*(s, \langle k, v\rangle)| \leq 1$.

Practically, this requirement is not limiting given some mechanism to uniquely identify each added value, e.g., by boxing them in tagged objects, and ensuring each added object receives a unique tag. Of course, an uncooperative implementation could thwart this scheme by changing its behavior when given tagged objects; this is excluded when implementations are *parametric*: if for every history h' admitted by an implementation Z, there exists an unambiguous history h such that $h' = \sigma(h)$ for some substitution σ. Intuitively the types of added values are completely opaque to parametric implementations, and we may assume that their histories are unambiguous.

Theorem 3.13. *The linearizability problem is solvable in polynomial time for unambiguous histories of collection types.*

Remark 3.14. We suspect that determining whether a given ADT is a collection type, and in particular determining reducibility, is generally undecidable. Nevertheless, since such determination is done once per ADT, and not per implementation, even manual proof is feasible. Once proved, the resulting polynomial time algorithm guaranteed by Theorem 3.13 is sound and complete for any implementation of the given ADT, thus avoiding the recurrent exponential cost.

The remaining sections prove this result by reduction to a tractable logical satisfiability problem.

4 OPERATION-ORDER SPECIFICATIONS

Although Gibbons and Korach [1997] prove that linearizability is NP-hard even for simple ADTs like the single-value register with read and write methods only, they also demonstrate that linearizability is solvable in polynomial time, for such registers, given a "reads-from" function mapping each read() $\Rightarrow v$ operation to the corresponding write(v) operation for a given instance of value v. While this reads-from function would be obvious were each value written at most once, multiple writes of the same value lead to ambiguity, giving rise to NP-hardness.

Exploiting this distinction, we refine the linearizability problem to one which avoids the inherent hardness of resolving ambiguity in the reads-from relation. Generalizing beyond registers, we consider a generic notion of history restriction, e.g., only histories in which any value is written

$$\underline{\text{Operation-Order Signature}}$$

$$\text{Operations} : \text{Sort}$$
$$\text{before} : \text{Operations} \times \text{Operations}$$

$$\underline{\text{Basic History Signature}}$$

$$\text{Operations, Methods, Values} : \text{Sort}$$
$$\text{method} : \text{Operations} \to \text{Methods}$$
$$\text{argument}_k : \text{Operations} \to \text{Values}$$
$$\text{result} : \text{Operations} \to \text{Values}$$
$$\text{before} : \text{Operations} \times \text{Operations}$$

Fig. 2. Signatures for operation-order specifications.

no more than once, and identify the conditions under which validating only the restricted set of histories is sound for establishing linearizability. In order to ensure that our refined linearizability problem is tractable, our problem input includes a logical specification of a given ADT in a logic for which we demonstrate a polynomial-time decision procedure, in Section 5. Sections 6 and 7 demonstrate that such sound logical specifications are guaranteed to exist for collection types.

Formally, the *operation-order signature* and *basic history signature* are the logical signatures containing the sorts and function symbols of Figure 2; a *history signature* is any signature extending the basic history signature. For a given signature Σ, a Σ-*formula* is a first order formula over Σ; we assume all formulas are in prenex normal form with matrix in conjunctive normal form (CNF). A Σ-*model* M is a finite Σ-structure such that $M(\text{before})$ is a partial order; M is *total* if $M(\text{before})$ is total, and otherwise *partial.* A Σ-model M' *extends* M when it extends M's order, i.e., $M'(\text{before}) \supseteq M(\text{before})$ and $M'(f) = M(f)$ for $f \neq \text{before}$.[2] A Σ-model M is *safe* for a formula F when $M' \models F$ for some total extension M' of M.

Example 4.1. The history formula

$$\forall x, y.\ (\text{method}(x) = \text{push} \wedge \text{method}(y) = \text{pop}$$
$$\wedge\ \text{argument}_1(x) = \text{result}(y)) \Rightarrow \text{before}(x, y)$$

dictates that push operations come before any pop operations which return their argument value.

For the basic history signature Σ, the *standard* Σ-*model* M_s of a given operation-label sequence s maps the Operations sort to the indices $\{0, \ldots |s| - 1\}$ of s, the Methods and Values sorts to the methods and values occurring in s, the before predicate to integer less-than (ordering the operations of s by their indices), and the method, argument_k, and result functions to functions f_m, $f_{\text{a},k}$, and f_r such that $f_\text{m}(i) = m$, $f_{\text{a},k}(i) = u_k$, and $f_\text{r}(i) = v$ where $s_i = m(\vec{u}) \Rightarrow v$.

A *history interpretation* I is a polynomial-time computable function mapping each history h to a history model $I(h)$ such that $I(h)(\text{before})(o_1, o_2)$ iff o_1 happens before o_2 in h. A history interpretation I is *sound* when for each history h and each linearization s of h, the standard model M_s extends $I(h)$. The *standard interpretation* I maps each history h to a Σ-model $I(h)$ mapping the method, argument_k, and result analogously to the standard model.

LEMMA 4.2. *The standard interpretation is sound.*

PROOF. Direct consequence of definitions. □

[2]Models that are equal up to operation renaming are considered equal.

A *history restriction* R is an onto binary relation over histories. An *R-history* is a history in the domain of R. For a given abstract data type, the restriction R is *sound* if h' is linearizable whenever h is, for all $\langle h, h' \rangle \in R$. For a given implementation Z, the restriction R is *sound* if for every history h' in Z, h' is included in the range of R and there exists an R-history h such that $\langle h, h' \rangle \in R$.

Example 4.3. Consider the *unambiguous restriction*, which relates each unambiguous history h — see Section 3 — to any history $\sigma(h)$ obtained by substitution. Intuitively, h is one possible disambiguation of $\sigma(h)$: operations adding the same values in $\sigma(h)$ add distinct values in h. Lemma 7.3 of Section 7 shows that the unambiguous restriction is sound for parametric ADTs and implementations, respectively.

The input to our refined linearizability problem thus includes a history formula specifying the ordering constraints among ADT operations, along with a history restriction.

Definition 4.4. For a given abstract data type, an *operation-order specification $S = \langle \Sigma, I, F, R \rangle$ of rank N* is a history signature Σ along with a history interpretation I and history formula F of quantifier rank at most N, and a sound history restriction R such that $I(h)$ is safe for F for all linearizable R-histories h.

Example 4.5. The basic history signature, the standard interpretation I, the history formula F of Example 4.1, and the unambiguous restriction of Example 4.3 constitute an operation-order specification for the stack type. Note that for linearizable unambiguous histories h, $I(h)$ is safe for F, since push operations are linearized before their matching pop operations in any stack-admitted linearization.

We say that an R-history h is OOS *linearizable* for a given operation-order specification $S = \langle \Sigma, I, F, R \rangle$ when $I(h)$ is safe for F. Note that any history that is not OOS linearizable is also not linearizable, by the "safety" constraint imposed by Definition 4.4. Completeness of OOS linearizability for determining linearizability thus holds.

Definition 4.6. The OOS*-linearizability problem* asks whether a given history is OOS linearizable for a given operation-order specification.

Example 4.7. The (sequential) history

$$\text{push}(1) \cdot \text{push}(2) \cdot \text{pop} \Rightarrow 1$$

is OOS linearizable with the specification of Example 4.5, as the push operation can be linearized before its matching pop. Of course, this history is not linearizable, since the pop operation does not return the most-recently pushed value.

The following notion of soundness ensures that linearizability of a given history can be deduced by establishing OOS linearizability. Thus given a sound operation-order specification of a given ADT, linearizability and OOS linearizability are equivalent over histories in the restriction.

Definition 4.8. An operation-order specification $S = \langle \Sigma, I, F, R \rangle$ for an abstract data type T is *sound* if any R-history which is OOS linearizable with S is linearizable to T.

Finally, whether validating only the restricted set of histories suffices to conclude the validation of all histories depends not only on the given ADT, but also on the particular ADT implementation under consideration.

THEOREM 4.9. *Let S be a sound operation-order specification for an abstract data type T with restriction R which is sound for a given implementation Z of T. Then Z is linearizable iff all R-histories of Z are* OOS *linearizable.*

Proof. For the "only-if" direction, let h be an R-history admitted by Z. Since h is linearizable by hypothesis, then h is also OOS linearizable by virtue that S is an operation-order specification.

For the "if" direction, let h be any history admitted by Z. If h happens to be an R-history, then by hypothesis it is OOS linearizable, and thus also linearizable by soundness of S.

Otherwise suppose h is not an R-history. Since R is sound for Z, h is included in the range of R and there exists some R-history h_R admitted by Z such that $\langle h_R, h\rangle \in R$. By hypothesis h_R is OOS linearizable, and thus also linearizable by soundness of S. Since $\langle h_R, h\rangle \in R$ and R is sound, then h is also linearizable. □

Section 7 demonstrates the existence of sound operation-order specifications for collection types by restriction to unambiguous histories, extension of the history signature and standard interpretation to include the operation predicates of Section 3, and leveraging the combinatorial properties of collection types found in Section 6 to derive history formulas.

5 POLYNOMIAL-TIME LINEARIZABILITY CHECKING

Since the linearizability problem for a given history and abstract data type is NP-complete [Gibbons and Korach 1997], it is polynomial-time equivalent to propositional satisfiability (SAT). Intuitively, the exponential enumeration of possible linearizations required to check linearizability in the worst case is akin to the exponential enumeration of possible satisfying assignments to check satisfiability. Recent work suggests approximating linearizability in polynomial time via possibly-unsound propositional reasoning which includes unit propagation yet excludes assignment speculation and backtracking [Emmi et al. 2015]. In this section we build on this intuition by demonstrating that this kind of reasoning is actually sound for deciding OOS linearizability. We demonstrate this via reduction to propositional Horn satisfiability, which neatly captures the absence of the expensive assignment speculation and backtracking required for satisfiability in general.

Definition 5.1. The *Horn satisfiability* problem asks whether a given set of propositional Horn clauses is satisfiable.

Lemma 5.2. *Horn satisfiability is solvable in linear time.*

Proof. See Dowling and Gallier [1984]. □

Our reduction from OOS linearizability to Horn satisfiability must overcome two key obstacles. First, history formulas are not propositional: they are first-order formulas containing function symbols, predicate symbols, and quantifiers. Second, deciding safety of a given history model involves constructing a total order from its happens-before partial order, and complete axiomatizations of total orders are not Horn since encoding totality requires non-Horn clauses such as $\forall x, y.\ x = y \lor p(x, y) \lor p(y, x)$, for a given order predicate p, to ensure that every two elements are related by p. In the following we approach these two obstacles, respectively, by first instantiating quantifiers and evaluating non-order related function symbols from the operations and labeling of a given history model, and then rewriting the resulting ground and quantifier-free operation-order formula so that applying the totality axiom at any subsequent reasoning step would be redundant. While the formula obtained through quantifier instantiation is an arbitrary propositional formula, which may contain other non-Horn clauses than the totality axiom, the rewriting step computes an equi-satisfiable Horn formula. Given a fixed maximum rank on history formulas, both steps are computable in polynomial time, and thus reduce OOS linearizability to Horn satisfiability.

F	$\text{inst}(F, M)$
$\forall x.F'$	$\bigwedge_{c \in C} \text{inst}(F'[c/x], M)$
$\exists x.F'$	$\bigvee_{c \in C} \text{inst}(F'[c/x], M)$
$\bigotimes_i F_i$	$\bigotimes_i \text{inst}(F_i, M)$
$\text{before}(t_1, t_2)$	$\text{before}(t_1, t_2)$
other atomic	true if $M + C \models F$ else false

where $C = M(\text{Operations}) \cup M(\text{Values})$.

Fig. 3. The instantiation of a formula F with model M defined recursively, where C is the finite set of operations and values occurring in M, $\bigotimes$ is an arbitrary logical connective, e.g., $\neg$, $\wedge$, $\vee$, and $M + C$ denotes the model obtained from M by adding identity interpretations for the constants C.

More technically, the *instantiation* of a history formula F with a history model M is the formula $\text{inst}(F \wedge \Theta, M) \wedge \text{ord}(M)$, where Θ are the axioms of total order on before:

$$\forall x, y, z.\ \text{before}(x, y) \wedge \text{before}(y, z) \Rightarrow \text{before}(x, z),$$
$$\forall x, y.\ x = y \vee \text{before}(x, y) \vee \text{before}(y, x), \text{ and}$$
$$\forall x, y.\ \neg\text{before}(x, y) \vee \neg\text{before}(y, x),$$

$\text{inst}(F \wedge \Theta, M)$ is defined by Figure 3, and $\text{ord}(M)$ is the conjunction of happens-before constraints from M:

$$\{\text{before}(o_1, o_2) : M(\text{before})(o_1, o_2)\}.$$

Essentially, $\text{inst}(F, M)$ replaces the quantified variables of F with the operations and values of M, and replaces atomic formulas unrelated to order with Boolean constants.

An operation-order formula is said to have a *total-order axiomatization* if it either:

- contains the above axioms Θ of total order on before, or
- is ground, quantifier-free, and contains the matrix Θ' of the above axioms instantiated via $\Theta'[o_1/x][o_2/y][o_3/z]$ for every triple of operation-sort constants o_1, o_2, o_3 occurring within [3].

Intuitively, when formulas contain total-order axiomatization we may treat the before predicate as uninterpreted since any satisfying model must map before to a total order.

The following lemmas demonstrate that instantiation is an effective reduction from the safety of a history model to the satisfiability of an operation-order formula.

Lemma 5.3. *The instantiation of a history formula F is a ground and quantifier-free operation-order formula with total-order axiomatization computable in polynomial time for any fixed quantifier rank formula F.*

Proof. The instantiation being a ground and quantifier-free operation-order formula with total-order axiomatization is a direct consequence of the definition. For a given history model M, it is computable in time $O(n^{max(k,3)} \cdot |F|)$ where n is the number of operations and values in M, $|F|$ is the size of the formula F, and k is its quantifier rank. □

Lemma 5.4. *The instantiation of a history formula F can be transformed to conjunctive normal form in polynomial-time for any fixed quantifier rank formula F.*

[3]These instantiations will contain only before atoms. The equality in the second axiom of Θ will be replaced by a boolean constant since operation-sort constants are interpreted to distinct operations.

Proof. The instantiation of a history formula F of quantifier rank k (assumed to be in CNF) contains at most $k + 2$ nestings of boolean operators which can be distributed to produce a CNF formula in time $O(n^{max(k,3)} \cdot |F|)$. □

Lemma 5.5. *The instantiation of a history formula F with history model M is satisfiable iff M is safe for F.*

Proof. A model M is safe for F iff there exists a total extension M_s of M such that $M_s \models F$. Since M_s differs with M just in the interpretation of before, and the interpretation of the latter in M_s is consistent with its interpretation in M, we have that $M_s \models \text{inst}(F \land \Theta, M) \land \text{ord}(M)$. Also, by definition, any model of $\text{inst}(F \land \Theta, M) \land \text{ord}(M)$ is a total extension of M (the total order axiomatization included in $\text{inst}(F \land \Theta, M)$ ensures that before is interpreted as a total order). Therefore, M is safe for F iff $\text{inst}(F \land \Theta, M) \land \text{ord}(M)$ is satisfiable. □

To further reduce the satisfiability of operation-order formulas to propositional Horn satisfiability, we suppose that formulas are in conjunctive normal form. Then let $\sim$ be the minimal equivalence relation on literals including $p(t_1, t_2) \sim \neg p(t_2, t_1)$ for any predicate p and terms $t_1 \neq t_2$. Intuitively, the $\sim$ relation relates literals that are equivalent under the assumption that p is an antisymmetric order with totality since $\neg p(t_2, t_1)$ would imply $p(t_1, t_2)$ and vice versa for terms $t_1 \neq t_2$. Extending $\sim$ to clauses, we relate two equal-length clauses $\bigvee \{l_0, \dots, l_n\} \sim \bigvee \{l'_0, \dots, l'_n\}$ when $l_i \sim l'_i$ for each $0 \leq i \leq n$. The *Hornification* of a quantifier-free operation-order formula $F = \bigwedge c_i$ in conjunctive normal form is the formula

$$\bigwedge \{c : \exists i.\ c \sim c_i \text{ and } c \text{ is Horn}\},$$

of all Horn clauses c related to some clause c_i of F.

Example 5.6. The Hornification of the non-Horn clause

$$\text{before}(o_1, o_2) \lor \text{before}(o_2, o_4) \lor \neg\text{before}(o_1, o_4),$$

is the following set of clauses,

$$\begin{gathered}\text{before}(o_1, o_2) \lor \neg\text{before}(o_4, o_2) \lor \neg\text{before}(o_1, o_4),\\ \neg\text{before}(o_2, o_1) \lor \text{before}(o_2, o_4) \lor \neg\text{before}(o_1, o_4),\\ \neg\text{before}(o_2, o_1) \lor \neg\text{before}(o_4, o_2) \lor \neg\text{before}(o_1, o_4),\\ \neg\text{before}(o_2, o_1) \lor \neg\text{before}(o_4, o_2) \lor \text{before}(o_4, o_1),\end{gathered}$$

which are in Horn form. All clauses are equivalent in theories where before is a total order.

The following lemmas demonstrate that Hornification is an effective reduction from satisfiability to Horn satisfiability of operation-order formulas.

Lemma 5.7. *The Hornification of a ground and quantifier-free operation-order formula in conjunctive normal form with total-order axiomatization is a ground, quantifier-free, and Horn operation-order formula.*

Proof. Follows by definition of Hornification. □

Lemma 5.8. *The Hornification of an operation-order formula is polynomial-time computable.*

Proof. Without loss of generality, we assume a fixed maximum clause length $k > 2$; any CNF formula can be rewritten to one with maximum clause length k in polynomial-time. In more details, every clause c of length m can be split into a conjunction of $O(m/k)$ clauses of length k that use $O(m/k)$ additional variables to ensure that one of the literals in c is true iff there exists an assignment for the additional variables such that all the k-length clauses are true. This transformation leads to an equivalent formula modulo the interpretation of the additional variables. Given a formula with n clauses of maximum clause length k, any given clause has $O(2^k)$ possible clauses related by $\sim$, and thus the number of clauses in the Hornification is limited by $O(n \cdot 2^k)$, which reduces to $O(n)$ for fixed k. □

The following result shows that the Hornification F' of a quantifier-free operation-order formula F with total-order axiomatization is equisatisfiable to F. We remark that these formulas are generally not equivalent, since F' may have models in which before is interpreted to a partial order, while F only has total models, since it has a total-order axiomatization.

Lemma 5.9. *The Hornification of a ground and quantifier-free operation-order formula F in conjunctive normal form with total-order axiomatization is satisfiable iff F is satisfiable.*

Proof. Since F is ground and quantifier-free, its satisfiability can be decided using a derivation based on the resolution rule of propositional logic (each atom is viewed as a boolean variable). We prove that any such derivation that uses as axioms clauses in F and derives a clause c can be rewritten to a derivation that uses as axioms clauses in the Hornification of F and derives a clause c' with $c \sim c'$. Instantiating this fact for c being the empty clause we get that F is satisfiable when its Hornification is satisfiable. The other direction is trivial since the clauses added in the Hornification of F can be always derived by the resolution rule from clauses in F.

W.l.o.g. we assume that all the clauses in F except those which express totality, i.e., the clauses of the form $\text{before}(o_1, o_2) \vee \text{before}(o_2, o_1)$, are Horn. Otherwise, every other non-Horn clause c can be replaced with some Horn clause c' such that $c' \sim c$. The obtained formula is equivalent to the original one.

Let $\overline{F}$ denote the Hornification of F, and $F \vdash c$ denote a derivation based on the resolution rule which uses as axioms clauses in F and derives the clause c. We prove that for every such formula F and every derivation $F \vdash c$,

$$\text{there exists } \overline{F} \vdash c' \text{ for all Horn clauses } c' \text{ with } c' \sim c. \tag{1}$$

We proceed by induction on the number N of non-Horn resolvents in $F \vdash c$. We write $F \vdash_N c$ for a derivation with N non-Horn resolvents.

Base case: A non-empty derivation $F \vdash_0 c$ may contain non-Horn clauses only as axioms, and the conclusion c is necessarily a Horn clause (we omit the case of empty derivations which is trivial). Consider an application of the resolution rule in this derivation with at least one non-Horn clause as premise (c_1 and c_2 are clauses and a is an atom):

$$\frac{c_1 \vee a \qquad c_2 \vee \neg a}{c_1 \vee c_2}$$

We call a the pivot of the resolution rule application.

By hypothesis, $c_1 \vee c_2$ is Horn and must contain exactly one positive literal b (if it contains no positive literal, then both premises are Horn). This positive literal must belong to c_1 (otherwise, both premises are Horn: $c_1 \vee a$ contains no positive literal other than a, and $c_2 \vee \neg a$ contains only b as a positive literal). Therefore, $c_1 \vee a$ is a clause expressing totality of the form $b \vee a$ with $b \sim \neg a$ and the derivation rule becomes:

$$\frac{b \vee a \qquad c_2 \vee \neg a}{b \vee c_2}$$

Since c_2 cannot contain other positive literal than b and $b \sim \neg a$, the conclusion $b \vee c_2$ of this rule is included in $\overline{F}$. Therefore, replacing all such resolution rule applications with axioms from $\overline{F}$, the derivation $F \vdash_0 c$ can be transformed to a derivation $\overline{F} \vdash c'$ for some $c' \sim c$.

To conclude, we prove that a derivation $\overline{F} \vdash c'$ for some Horn clause $c' \sim c$ can be rewritten to a derivation $\overline{F} \vdash c''$ for every Horn clause $c'' \sim c'$. When c'' is obtained from c' by replacing a positive literal a with $\neg b$ such that $a \sim \neg b$, the same transformation can be applied to all the resolvents in the derivation $\overline{F} \vdash c'$ and obtain a new derivation $\overline{F} \vdash c''$ (a Horn clause remains Horn when a positive literal is replaced by a negative one). Next, w.l.o.g., assume that c'' is obtained from c' by replacing a positive literal a with $\neg b$ and a negative literal $\neg a'$ with b' such that $a \sim \neg b$ and $\neg a' \sim b'$. We proceed by induction on the number of applications of the resolution rule in $\overline{F} \vdash c'$. W.l.o.g., assume that the last application of the resolution rule is of the form:

$$\frac{a \vee c_1 \vee \neg d \qquad \neg a' \vee c_2 \vee d}{a \vee c_1 \vee \neg a' \vee c_2}$$

where c_1 and c_2 are clauses that contain only negative literals. This rule can be rewritten as follows

$$\frac{\neg b \vee c_1 \vee d' \qquad b' \vee c_2 \vee \neg d'}{\neg b \vee c_1 \vee b' \vee c_2}$$

where $\neg d' \sim d$. Applying also the induction hypothesis, we get that there exists a derivation $\overline{F} \vdash c''$.

Induction step: Assume that (1) holds for all derivations with N non-Horn resolvents, and let $F \vdash_{N+1} c$ be a derivation. Let

$$\frac{c_1 \vee a \qquad c_2 \vee \neg a}{c_1 \vee c_2}$$

be an application of the resolution rule in $F \vdash_{N+1} c$ such that all the resolvents in the derivation of the premises $c_1 \vee a$ and $c_2 \vee \neg a$ are Horn and $c_1 \vee c_2$ is not Horn. The latter derivation may still contain other non-Horn clauses than the conclusion, but they are necessarily axioms.

One of the premises above must be a resolvent (since the only non-Horn axioms are those expressing totality, and two such axioms can't be the premises of the same derivation rule). Also, remark that $c_1 \vee a$ and $c_2 \vee \neg a$ can't be both Horn clauses (otherwise, $c_1 \vee c_2$ would be Horn). Also, the non-Horn clause between the two is necessarily an axiom by the choice of the resolution rule application. Therefore, $c_1 \vee a$ is an axiom expressing totality of the form $b \vee a$ with $b \sim \neg a$ and the derivation rule becomes:

$$\frac{b \vee a \qquad c_2 \vee \neg a}{b \vee c_2}$$

If b is not the pivot of a further resolution rule application, then this rule is vacuous and it can be eliminated. More precisely, all the further applications of the resolution rule are enabled if $c_2 \vee \neg a$ is considered instead of $b \vee c_2$. Eliminating this application will lead to a derivation with at most N non-Horn resolvents for which the induction hypothesis holds.

Assume now that b is the pivot of a further resolution rule application. The derivation $F \vdash_{N+1} c$ can be then outlined as (Ax_i with $i \in [1, 4]$ are sets of axioms):

$$
\dfrac{
  \dfrac{
    \dfrac{
      \begin{array}{c}\dfrac{b \vee a \qquad \dfrac{\begin{array}{c}\underline{Ax_1}\\ \vdots\end{array}}{c_2 \vee \neg a}}{c_2 \vee b}\\ \vdots\end{array}
      \qquad
      \begin{array}{c}\underline{Ax_2}\\ \vdots\end{array}
    }{c_3 \vee b}
    \qquad
    \dfrac{\begin{array}{c}\underline{Ax_3}\\ \vdots\end{array}}{c_4 \vee \neg b}
  }{c_3 \vee c_4}
  \qquad
  \begin{array}{c}\underline{Ax_4}\\ \vdots\end{array}
}{c}
$$

Applying the induction hypothesis for the derivations $Ax_1 \vdash_0 c_2 \vee \neg a$ and $Ax_3 \vdash_{N'} c_4 \vee \neg b$ with $N' \leq N$, we get that there exist derivations $\overline{Ax_1} \vdash c_2' \vee \neg a$ and $\overline{Ax_3} \vdash c_4' \vee a$ for some Horn clauses $c_2' \vee \neg a \sim c_2 \vee \neg a$ and $c_4' \vee a \sim c_4 \vee \neg b$. The derivation $F \vdash_{N+1} c$ can be transformed as follows:

$$
\dfrac{
  \dfrac{
    \begin{array}{c}\dfrac{\dfrac{\begin{array}{c}\underline{\overline{Ax_1}}\\ \vdots\end{array}}{c_2' \vee \neg a} \qquad \dfrac{\begin{array}{c}\underline{\overline{Ax_3}}\\ \vdots\end{array}}{c_4' \vee a}}{c_2' \vee c_4'}\\ \vdots\end{array}
    \qquad
    \begin{array}{c}\underline{\overline{Ax_2}}\\ \vdots\end{array}
  }{c_3' \vee c_4'}
  \qquad
  \begin{array}{c}\underline{\overline{Ax_4}}\\ \vdots\end{array}
}{c'}
$$

This transformed derivation is correct because of the following:

- $\{c_2 \vee b\} \cup Ax_2 \vdash_{N'} c_3 \vee b$ with $N' \leq N$ implies $\{c_2\} \cup Ax_2 \vdash_{N'} c_3$. Then, by the induction hypothesis, there exists a derivation $\overline{\{c_2\}} \cup \overline{Ax_2} \vdash c_3'$ for some $c_3' \sim c_3$.
- since increasing the set of literals in the premises doesn't disable any resolution rule application, we get that there exists a derivation $\overline{\{c_2 \vee c_4'\}} \cup \overline{Ax_2} \vdash c_3' \vee c_4'$.
- for simplicity, assume that $c_2' \vee c_4'$ is the only axiom in the latter derivation which belongs to $\overline{\{c_2 \vee c_4'\}}$. We get that $\overline{Ax_1} \cup \overline{Ax_2} \cup \overline{Ax_3} \vdash c_3' \vee c_4'$. If the simplifying assumption doesn't hold, then the derivation of $c_3' \vee c_4'$ would contain derivations of other Horn formulas in $\overline{\{c_2 \vee c_4'\}}$. These derivations can be obtained trough a trivial rewriting from the derivation of $\overline{Ax_1} \cup \overline{Ax_3} \vdash c_2' \vee c_4'$.
- finally, since $\{c_3 \vee c_4\} \cup Ax_4 \vdash_{N'} c$ with $N' \leq N$ we get that there exists a derivation $\overline{\{c_3 \vee c_4\}} \cup \overline{Ax_4} \vdash c'$ for some $c' \sim c$. Again, we can assume for simplicity that $c_3' \vee c_4'$ is the only axiom in these derivations which belongs to $\overline{\{c_3 \vee c_4\}}$. Otherwise, we can proceed as in the previous case.

This transformed derivation has axioms from $\overline{F}$ and derives a Horn clause $c' \sim c$. Such a derivation can be rewritten to a derivation $\overline{F} \vdash c''$ for every Horn clause $c'' \sim c'$. This concludes our proof. □

The proof of Lemma 5.9 relies on a stronger property of the Hornification: any Horn clause c derived in a resolution proof from the original formula F can be also derived from the Hornification of F. (The reverse holds trivially since any clause added to the Hornification can be derived from the clauses of F.) When c is the empty clause, this property amounts to equisatisfiability. Looking at the first steps in a derivation of c from F, a resolution rule application that uses as a premise an instance of the totality axiom corresponds to replacing an atom $\neg p(t_1, t_2)$ with $p(t_2, t_1)$. This transformation is in principle anticipated with the new clauses added in the Hornification of F. However, the restriction to Horn clauses may force other atoms to be replaced with equivalent counterparts. Therefore, replacing that resolution rule application with a clause from the Hornification may affect other parts of the derivation as well. Nevertheless, these parts of the derivation can be rewritten using again clauses from the Hornification.

Combining instantiation, Hornification, and propositional Horn satisfiability yields an efficient procedure for OOS linearizability. Its asymptotic complexity is dominated by quantifier instantiation which is $O(n^k)$, where k is the rank of the input operation-order signature.

THEOREM 5.10. OOS *linearizability is solvable in polynomial time for fixed-rank operation-order signatures.*

PROOF. Given an operation-order specification $\langle \Sigma, I, F_0, R \rangle$ and a history h, from the definitions of history interpretation and operation-order specification we obtain a history model $I(h)$ in polynomial time which is safe with F_0 iff h is OOS linearizable. By Lemmas 5.3–5.9, we know that $I(h)$ is safe with F_0 iff the polynomial-time instantiation F_1 of F_0 with $I(h)$ is satisfiable, which in turn is satisfiable iff the polynomial-time Hornification F_2 of F_1 is satisfiable. Constructing a propositional Horn formula F_3 from F_2 by replacing each atom $\mathrm{before}(\mathrm{o}_1, \mathrm{o}_2)$ with a propositional variable $x_{\mathrm{o}_1,\mathrm{o}_2}$, it follows that F_3 is equisatisfiable to F_2, and by Lemma 5.2, its satisfiability is polynomial-time computable. □

The remaining sections demonstrate that OOS linearizability amounts to a sound decision procedure for linearizability of collection types.

6 COMPLETE COLLECTION-VIOLATION PATTERNS

In this section we demonstrate the fundamental combinatorial properties enjoyed by collection types enabling our reduction to OOS linearizability. Essentially, we show that any violation to a collection in an unambiguous sequence (see Section 3) can always be reduced to one of a finite number of violation patterns. More technically, we characterize violations via an embedding relation between unambiguous sequences, and demonstrate that this relation always yields a bounded set of minimal sequences of which every violation embeds at least one. Later, in Section 7, we demonstrate that this minimal set of violation patterns defines the history formula of an operation-order specification against which the linearizability of a given sequence is judged.

Our embedding relation is built from three smaller relations $\rightarrow_{\mathrm{s}}$, $\rightarrow_{\mathrm{r}}$, and $\rightarrow_{\mathrm{v}}$, relating operation-label sequences s_1 and s_2 when s_2 is obtained from s_1 by,

- applying a substitution ($\rightarrow_{\mathrm{s}}$),
- removing a read-only operation ($\rightarrow_{\mathrm{r}}$), or
- removing all operations which touch a given value ($\rightarrow_{\mathrm{v}}$).

Our embedding relation $\sqsupseteq$ is the smallest reflexive and transitive binary relation containing $\rightarrow_{\mathrm{s}}$, $\rightarrow_{\mathrm{r}}$, and $\rightarrow_{\mathrm{v}}$. The embedding relates $s_1 \sqsupseteq \sigma(s_2)$ for any substitution σ, where s_2 is obtained from s_1 by removing all operations touching any given set of values, and any number of read-only operations. Violations to a collection are characterized by this embedding relation in the sense that any unambiguous sequence s_1 which embeds a violation s_2 (i.e., $s_1 \sqsupseteq \sigma(s_2)$) is also a violation (see Lemma 6.6). For instance, adding read-only operations to a violation s_2 results in a sequence s_1 which is also a violation.

Example 6.1. The embedding relation $\sqsupseteq$ relates the following unambiguous sequences of decreasing length

$$\begin{aligned}
&\text{push}(1) \cdot \text{push}(2) \cdot \text{push}(3) \cdot \text{pop} \Rightarrow 3 \cdot \text{peek} \Rightarrow 2 \cdot \text{pop} \Rightarrow 1 \\
\rightarrow_{\text{v}} \quad & \text{push}(1) \cdot \text{push}(2) \cdot \text{peek} \Rightarrow 2 \cdot \text{pop} \Rightarrow 1 \\
\rightarrow_{\text{r}} \quad & \text{push}(1) \cdot \text{push}(2) \cdot \text{pop} \Rightarrow 1 \\
\rightarrow_{\text{s}} \quad & \text{push}(3) \cdot \text{push}(4) \cdot \text{pop} \Rightarrow 3,
\end{aligned}$$

neither of which are admitted by the stack type.

We show that the embedding relation $\sqsubseteq$ has a finite set of minimal elements when restricted to the set of sequences that touch a bounded set of values and where the number of mutators (non-readonly operations) that touch a given value is also bounded. This property is captured formally with wqos: a *well-quasi-ordering (wqo) Q* on a set X is a reflexive, transitive binary relation on X for which in every infinite sequence $x_0 x_1 \dots$ of elements from X, there exists indices $i < j$ such that $Q(x_i, x_j)$.

For a local type, a value v is *k-valent* in a sequence s when $|\text{touches}^*(s, v) \cap \neg\text{readonly}^*(s)| \leq k$, and a sequence s is *k-valent* when all values are k-valent in s. A sequence s is *k-valued* if at most k unique input or output parameters occur within, i.e., $| \biguplus_i \text{vals}(s_i)| \leq k$. We write V_k to denote the set of sequences which are both k-valent and k-valued.

Example 6.2. All the sequences in Example 6.1 are 2-valent since the only mutators touching a given value are push and pop. Moreover, they all belong to V_3 since they are 3-valued.

Lemma 6.3. *$\sqsubseteq$ is a wqo on V_k for all local types and $k \in \mathbb{N}$.*

Proof. Let $(s_i)_{i \in \mathbb{N}}$ be an infinite sequence[4] of sequences from V_k. Since V_k is closed under value substitution, there exists a set of values V with $|V| \leq k$ and conflict-free substitutions σ_i with range V such that $s'_i = \sigma_i(s_i) \in V_k$. Let μ be a map from sequences s to multisets $\mu(s) = \{s(j) : j \in (\neg\text{readonly})^*(s)\}$ of non-readonly labels. Since T is local, the labels of the s'_i sequences come from a finite set. Moreover, the s'_i sequences are k-valent which implies that the range of possible multisets $\mu(s'_i)$ is bounded. It follows that there exists an infinite subsequence $(s'_i)_{i \in I}$ for $I \subseteq \mathbb{N}$ of operation-label sequences with the same mutator multiset, i.e., $\mu(s'_{i_1}) = \mu(s'_{i_2})$ for all $i_1, i_2 \in I$. By Higman's Lemma, the subsequence order is a wqo on sequences from a bounded interface, which guarantees the existence of $i_1, i_2 \in I$ such that $i_1 < i_2$ and s'_{i_1} is a subsequence of s'_{i_2}. Since the mutators of s'_{i_1} and s'_{i_2} are identical, it follows that $s'_{i_1} = s'_{i_2}$ or s'_{i_1} is obtained from s'_{i_2} by removing only read-only operations. By the definition of $(s'_i)_{i \in \mathbb{N}}$, we get that $\sigma_{i_1} \circ \sigma^{-1}_{i_2}(s'_{i_1})$ is a subsequence of s'_{i_2} and that $\sigma_{i_1} \circ \sigma^{-1}_{i_2}(s'_{i_1})$ is obtained from s'_{i_2} by removing only read-only operations. Thus $s_{i_1} \sqsubseteq s_{i_2}$. □

Given an abstract data type T, we write P_k to denote a minimal set of $\sqsubseteq$-minimal sequences representing $\bar{T} \cap V_k$, i.e., such that $s \in (V_k \setminus T)$ iff there exists $s' \in P_k$ with $s' \sqsubseteq s$. Intuitively, any violation to the data type T that belongs to V_k will embed one of the sequences in P_k, referred to also as violation patterns.

[4]To avoid confusion here we write s_i to denote the ith operation-label sequence in a sequence $s_0 s_1 \dots$ of operation-label sequences, and $s(j)$ to denote the jth operation label of an operation-label sequence s.

Example 6.4. If T is the stack type with push and pop operations, then P_2 consists of

$$\begin{aligned}&\mathrm{pop} \Rightarrow 1\\&\mathrm{pop} \Rightarrow 1 \cdot \mathrm{push}(1)\\&\mathrm{push}(1) \cdot \mathrm{pop} \Rightarrow 1 \cdot \mathrm{pop} \Rightarrow 1\\&\mathrm{push}(1) \cdot \mathrm{push}(2) \cdot \mathrm{pop} \Rightarrow 1\\&\mathrm{push}(1) \cdot \mathrm{push}(2) \cdot \mathrm{pop} \Rightarrow 1 \cdot \mathrm{pop} \Rightarrow 2\end{aligned}$$

By Lemma 6.3 the set P_k is finite. It can be effectively computed by tracking minimal elements within sequences in V_k over a fixed set of k values and with at most one readonly operation[5] whose number is bounded by $k^2 + 1$ (there are at most k mutators that can touch one of the k values).

Corollary 6.5. *P_k contains at most $k^2 + 1$ elements and it is computable.*

For the remainder we fix a minimal P_k. We show that any sequence that embeds one of the patterns in P_k, for any k, is a violation to a local and parametric abstract data type. To this, we introduce a weakening of locality, called *partitionability*, that is enough to conclude this fact (together with parametricity).

An abstract data type T is *partitionable* when $s' \in T$ whenever $s \in T$ for some $s' \sqsubseteq s$. Note that any partitionable type is also parametric since $\sigma(s) \sqsubseteq s$ for all s.

Lemma 6.6. *Local parametric types are partitionable.*

Proof. Let T be a local parametric type, s a sequence in T, and $s' \sqsubseteq s$ for some sequence s'. W.l.o.g., we assume that $s' \rightarrow_{\mathrm{s}} s$, $s' \rightarrow_{\mathrm{r}} s$ or $s' \rightarrow_{\mathrm{v}} s$. The case $s' \rightarrow_{\mathrm{s}} s$ follows directly from the fact that T is parametric.

If $s' \rightarrow_{\mathrm{r}} s$, then there exist sequences s_1 and s_2, and operation label ℓ, such that $s = s_1 \cdot \ell \cdot s_2$, $s' = s_1 \cdot s_2$, and readonly$(s_1 \cdot \ell)$ holds. By the definition of the readonly predicate, $s_1 \cdot \ell$ and s_1 are equivalent and thus, $s_1 \cdot \ell \cdot s_2 \in T$ iff $s_1 \cdot s_2 \in T$. Therefore, $s' \in T$.

When $s' \rightarrow_{\mathrm{v}} s$, the fact that $s' \in T$ can be proved by induction on the length of s. By definition, s' is obtained from s by removing the labels on positions $J = \mathrm{touches}^*(s, v)$ for some value v. Essentially, there are two cases two consider.

case 1: If s ends with a label that touches v, i.e., $|s| - 1 \in J$, then s without that label is still admitted by T (by prefix closure) and the induction hypothesis applies.

case 2: Otherwise, we have $s = s_1 \cdot \ell$ for some sequence s_1 and label ℓ, such that $\neg\mathrm{uses}(s_1 \cdot \ell, v)$. Therefore, $\mathrm{drop}(s_1, v) \cdot \ell \in T$. Since T is local, we have that $v \notin \mathrm{vals}(\ell)$ which implies $\mathrm{drop}(s_1, v) \cdot \ell = \mathrm{drop}(s, v)$, and that $J = \{i : v \in \mathrm{vals}(s_i)\}$ which implies $\mathrm{drop}(s, v) = s'$. Hence, $s' \in T$. □

Let E_k be the set

$$E_k = \{s : \exists s' \in P_k.\, s' \sqsubseteq s\}$$

of sequences which embed one of the patterns in P_k. When the abstract data type T is parametric and partitionable, E_k contains only sequences which are not admitted by T.

Lemma 6.7. *If T is parametric and partitionable and $s \in E_k$ then $s \notin T$.*

Proof. Given a sequence $s \in E_k$ there exists a $\sqsubseteq$-minimal sequence $s' \in P_k$ such that $s' \sqsubseteq s$, and by definition of $\sqsubseteq$ it follows that $s' = \sigma(s'')$ for some subsequence s'' of s and substitution σ. By definition of P_k we know $\sigma(s'') \notin T$, and then $s' \notin T$ since T is parametric by hypothesis. Finally $s \notin T$ since T is partitionable and $s'' \sqsubseteq s$ by definition of $\sqsubseteq$. □

[5]We use the fact that a correct sequence cannot become a violation by adding a readonly operation.

We strengthen the result in Lemma 6.7 by showing that there exists some k such that E_k contains *exactly* all the *unambiguous* sequences which are k-valent and violations to a given collection. Proving also that all the *unambiguous* sequences which are not k-valent for some $k \geq 2$ are violations to a given collection leads to a characterization of unambiguous violations that is describable using a history formula.

A set of sequences X is *V_k-reducible* if $s \in X$ whenever s is k-valent and $s' \in X$ for all $s' \in V_k$ such that $s' \sqsubseteq s$.

Lemma 6.8. *The set of unambiguous sequences admitted by a collection are* 2*-valent.*

Proof. Let s be a unambiguous sequence admitted by a collection T. Because T is value-invariant, for any label ℓ in s such that $\text{touches}(\ell, v) \wedge \neg\text{readonly}(\ell)$ holds for some value v, we have that $\text{adds}(\ell, v) \vee \text{removes}(\ell, v)$ holds. Then, since s is unambiguous, there exists at most one label ℓ such that $\text{adds}(\ell, v)$ holds. This implies that there exists at most one label satisfying $\text{removes}(\ell, v)$, and thus s is 2-valent. □

Lemma 6.9. *The set of unambiguous sequences admitted by a collection is V_k-reducible for some $k \geq 2$.*

Proof. Let s be a unambiguous sequence, i.e., $\text{adds}(\ell, v)$ holds for at most one label ℓ in s, for each value v, and T a collection. Lemma 6.8 shows that $s \in T$ implies that s is 2-valent. Then, since T is reducible, there exists some bound k such that $s \in T$ whenever $\text{keep}(s, V) \in T$ for every subset $V \subseteq \text{vals}(s)$ of size $|V| \leq k$. W.l.o.g., assume that $k \geq 2$. Indeed, by definition, if T is reducible with some bound k then it is reducible with any greater bound $k' \geq k$. We have that $\text{keep}(s, V) \sqsubseteq s$ because T is local, i.e., every operation touches exactly the values appearing in its label. Moreover, if s is k-valent, then all the subsequences $\text{keep}(s, V)$ are k-valent which concludes the proof. □

A local abstract data type T has *bounded violations in unambiguous sequences* when there exists some bound $k \geq 2$ such that for every unambiguous sequence s, we have that $s \notin T$ iff s is not k-valent or $s \in E_k$.

Theorem 6.10. *Every collection T has bounded violations in unambiguous sequences.*

Proof. We have to show that for every unambiguous sequence s, $s \notin T$ iff s is not k-valent or $s \in E_k$. The "if" direction holds from Lemmas 6.6, 6.7, and 6.8. For the "only-if" direction, Lemma 6.9 implies that the set of unambiguous sequences is V_k-reducible, for some k. Therefore, for any unambiguous sequence $s \notin T$, either s is not k-valent, or there exists a sequence $s' \in V_k$ with $s' \sqsubseteq s$ such that $s' \notin T$. Since $s' \in (\bar{T} \cap V_k)$ there exists a $\sqsubseteq$-minimal sequence $s'' \in P_k$ such that $s'' \sqsubseteq s'$, and thus $s'' \sqsubseteq s$ because $\sqsubseteq$ is transitive. Then $s \in E_k$ by definition of E_k. □

The next section uses the result in Theorem 6.10 to define a history formula that describes precisely the unambiguous sequences admitted by a collection, which is the essential ingredient of a sound operation-order specification.

7 SOUND OPERATION-ORDER SPECIFICATIONS

In this section we leverage the combinatorial properties of Section 6 to demonstrate that collection types have sound operation-order specifications. We construct these specifications automatically given a set of violation patterns and a local interpretation for the operation predicates of a given collection type — both which could be inferred using Emmi and Enea [2016]'s algorithm for extracting patterns from execution samples. The specifications use a restriction to unambiguous histories, and extend the basic history signature with the operation predicates of collection types. The extended signature is interpreted using the local interpretation for a given type, and history

Extended History Signature

...
readonly : Operations
adds : Operations × Values
uses : Operations × Values

Fig. 4. The Extended History Signature includes predicate and function symbols of Figure 2's Basic History Signature.

formulas are essentially instantiations of a template stating that the violation patterns of a given type are not embedded in a given linearization.

Technically, the extended history signature of Figure 4 extends the basic history signature of Section 4 with additional symbols for the operation predicates of Figure 1. The *(extended) standard model* M_s of a given operation-label sequence s for the extended history signature extends the standard model for the basic history signature by interpreting each operation-predicate symbol P according to the meta-theory predicate P, i.e., $M_s(\mathsf{P})(i, \vec{v})$ iff $P(s_0 \dots s_i, \vec{v})$.

We extend the standard history interpretation of Section 4 to the extended signature by mapping each logical operation predicate symbol P to $P(O(o), \vec{v})$ where P is the local interpretation of the meta-theory predicate corresponding to P, and O is the operation-labeling function of the given history $\langle O, < \rangle$. Intuitively, this extended standard interpretation which ignores operation context is sufficient for local types because the labels alone determine the interpretation.

Lemma 7.1. *The extended standard interpretation is sound for local abstract data types.*

Proof. This follows easily by combining Lemma 4.2 with the fact that operation predicates are evaluated identically, using only the final operation label, in each linearization of a given history when the given type is local. □

Next, we construct a template which yields a given collection type's history formula given the set P_k of minimal violation patterns of a k-reducible collection type. Let $f : V \to |V|$ be a function mapping the set V of values used in a given operation-label sequence to natural numbers. The *operation formula* L_i for the ith operation $s_i = m(\vec{v}) \Rightarrow v$ of the sequence s is the formula

$$L_i = \mathsf{method}(x_i) = m \wedge \bigwedge_j \mathsf{argument}_j(x_i) = y_{f(v_j)}$$
$$\wedge\ \mathsf{result}(x_i) = y_{f(v)} \wedge \bigwedge_{i<j} \mathsf{before}(x_i, x_j)$$

with free operation-sort variables $\vec{x}$ and value-sort variables $\vec{y}$ dictating that operation x_i has the same label and operation-order as s_i.

For operation-label sequences s we construct the *sequence formula* F_s as the ground formula

$$\exists \vec{x}, \vec{y}.\ \bigwedge_i L_i \wedge \mathsf{distinct}(\vec{x}) \wedge \mathsf{distinct}(\vec{y})$$
$$\wedge\ \forall x.\ \bigvee_i x = x_i \vee \mathsf{readonly}(x) \vee \neg \bigvee_i \mathsf{touches}(x, y_i)$$

where $\mathsf{distinct}(\vec{x})$ is the conjunction of disequalities $x_i \neq x_j$ for $i < j$. The sequence formula F_s labels the quantified operations $\vec{x}$ and values $\vec{y}$ according to s, imposing that variables $\vec{x}$ and $\vec{y}$ are interpreted to distinct elements in their domain, and insisting that all other operations are either read-only or touch distinct values from those appearing in s.

The *bounded valence formula* G_k encodes the presence of a value for which a given sequence is not k-valent,

$$G_k = \exists v, x_0, \dots, x_k . \bigwedge_i \neg \mathsf{readonly}(x_i) \land \mathsf{touches}(x_i, v),$$

for operation-sort variables x_i and value-sort variable v. The *embedding formula* F_k encodes the embedding of the $\sqsubseteq$-minimal sequences P_k or the violation of k-valence:

$$F_k = \Big(\bigvee_{s \in P_k} F_s \Big) \lor G_k$$

The following establishes correspondence of E_k and F_k.

Lemma 7.2. *$M_s \models F_k$ iff $s \in E_k$ or s is not k-valent.*

Proof. For a sequence $s \in P_k$ and another sequence s', the standard model $M_{s'}$ satisfies F_s iff s is obtained from s' by removing readonly operations or sets of operations that touch some value (expressed by the sub-formula of F_s that follows the quantifier $\forall x$), or applying some substitution σ (since values are existentially-quantified). Therefore, $M_{s'} \models F_s$ iff $s \sqsubseteq s'$. Therefore, $M_{s'} \models F_k$ for some sequence s' iff s is not k-valent or there exists a $\sqsubseteq$-minimal element s with $s \sqsubseteq s'$, which is equivalent to $s' \in E_k$. □

Finally, we leverage the restriction to unambiguous sequences given by Example 4.3 in Section 4; we repeat its definition here for clarity. The *unambiguous restriction* relates each unambiguous history h to any history $\sigma(h)$ obtained by substitution. Each unambiguous history h is related to many unambiguous histories: one $\sigma(h)$ per choice of σ. Given an ambiguous history $\sigma(h)$ we can think of h as one possible disambiguation of the operations of $\sigma(h)$ such that operations adding the same values in $\sigma(h)$ add distinct values in h.

Lemma 7.3. *The unambiguous restriction is sound for parametric types and parametric implementations.*

Proof. Consider a pair $\langle h, h' \rangle$ in the unambiguous restriction with $h' = \sigma(h)$. On the one hand, if h is linearizable to some sequence s of a parametric ADT, then applying the same linearization order on h' yields a sequence s' which must be admitted since $s' = \sigma(s)$; thus soundness holds. On the other hand, if h' is admitted by a given parametric implementation, then h is also admitted by definition; thus soundness holds. □

Combining these elements together we arrive at the soundness of our reduction to OOS linearizability.

Theorem 7.4. *There exists a sound operation-order specification for any collection type.*

Proof. Let $k \geq 2$ be a natural number by which the given abstract data type T is reducible, or arbitrary otherwise. We construct an operation-order specification $S = \langle \Sigma, I, F, R \rangle$ for T as follows:

- Σ is the extended history signature,
- I is the standard interpretation for Σ,
- $F = \neg F_k$ is the embedding formula negated [6], and
- R is the unambiguous restriction (defined in Example 4.3).

[6]The matrix of F can be transformed to CNF in linear time.

We prove that S is indeed an operation-order specification for T. The unambiguous restriction R is sound for T, by Lemma 7.3, since T is parametric. Then, let h be a linearizable unambiguous history. By definition, the set of unambiguous sequences in T contains a linearization s of h. Since T is local and parametric, then by Lemmas 6.6 and 6.7, $M_s \models F$. By Lemma 7.1, the standard interpretation I is sound and thus, $I(h)$ is safe for F.

To prove the soundness of S, let h be an unambiguous history such that $I(h)$ is safe for F. Then, there exists a total model M_s extending $I(h)$ such that $M_s \models F$. If T is also value-invariant and reducible, then by Lemma 7.2 and Theorem 6.10, we get that s is an unambiguous sequence in T. Since M_s extends $I(h)$, we have that s is a linearization of h and thus, h is linearizable. □

This concludes our reduction to OOS linearizability.

8 RELATED WORK

While several static techniques have been developed to prove linearizability [Abdulla et al. 2013; Amit et al. 2007; Bouajjani et al. 2015b; Dodds et al. 2015; Dragoi et al. 2013; Henzinger et al. 2013; Herlihy and Wing 1990; Khyzha et al. 2016; Liang and Feng 2013; Liu et al. 2009; O'Hearn et al. 2010; Schellhorn et al. 2012; Sergey et al. 2015a,b; Shacham et al. 2011; Vafeiadis 2010; Zhang 2011], few have addressed dynamic techniques such as testing and runtime verification. Early work by Wing and Gong [1993] proposed an exponential-time monitoring algorithm for arbitrary abstract data types. This algorithm was later optimized by Lowe [2006], and further optimized by Horn and Kroening [2015]; neither avoided exponential-time asymptotic complexity. Burckhardt et al. [2010] and Burnim et al. [2011] implement exponential-time monitoring algorithms in their tools for bounded testing of concurrent data structures in .NET and Java, and Zhang et al. [2015] implement a similar algorithm in their tool for testing "quasi-linearizability."

Bouajjani et al. [2015b] propose an unsound polynomial-time monitoring algorithm which admits false negatives due to operation-order abstraction. While Emmi et al. [2015] propose a polynomial-time algorithm similar to ours, and measure its practical soundness by empirical comparison with sound exponential-time algorithms, they provide no argument for theoretical soundness. While Bouajjani et al. [2015a] discover combinatorial properties of collection types related to ours, i.e., using the locality of collection types, their methods demonstrate decidable static verification, i.e., for all executions of a given implementation; their result does not yield polynomial-time algorithms for checking individual executions.

From the complexity standpoint, Gibbons and Korach [1997] showed that monitoring even the single-value register type is NP-hard in ambiguous histories, yet becomes polynomial-time for unambiguous histories. As far as we are aware, this is the only previously-known case of polynomial-time monitoring for linearizability. Alur et al. [2000] 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 [2015] established EXPSPACE-completeness. Bouajjani et al. [2013] showed that the problem becomes undecidable once the number of concurrent operations is unbounded.

9 CONCLUSION

We have demonstrated a sound, complete, and tractable linearizability-monitoring algorithm for collection types. As far as we know, this is the first result of its kind beyond the single-value register case established by Gibbons and Korach [1997]. Since tractable monitoring enables testing and runtime verification, both practical means to error detection, our result fosters increased implementation of concurrent data structures, perhaps even by non-experts. Our result can also guide the design of abstract data types which admit efficient monitors, by respecting the value-invariance, locality, parametricity, and reducibility criteria.

While theoretical in nature, our result is directly relevant to practice, essentially establishing the soundness of Emmi et al. [2015]'s efficient monitor implementation. Furthermore, the local interpretations of Section 3 and violation patterns of Section 6, which are needed to derive a monitor for a given collection type, can be automatically inferred, e.g., using Emmi and Enea [2016]'s algorithm for extracting patterns from execution samples.

ACKNOWLEDGMENTS

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).

REFERENCES

Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezine. 2013. An Integrated Specification and Verification Technique for Highly Concurrent Data Structures. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science)*, Nir Piterman and Scott A. Smolka (Eds.), Vol. 7795. Springer, 324–338. https://doi.org/10.1007/978-3-642-36742-7_23

Rajeev Alur, Kenneth L. McMillan, and Doron A. Peled. 2000. Model-Checking of Correctness Conditions for Concurrent Objects. *Inf. Comput.* 160, 1-2 (2000), 167–188. https://doi.org/10.1006/inco.1999.2847

Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. 2007. Comparison Under Abstraction for Verifying Linearizability. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings (Lecture Notes in Computer Science)*, Werner Damm and Holger Hermanns (Eds.), Vol. 4590. Springer, 477–490. https://doi.org/10.1007/978-3-540-73368-3_49

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2013. Verifying Concurrent Programs against Sequential Specifications. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 290–309. https://doi.org/10.1007/978-3-642-37036-6_17

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2015a. On Reducing Linearizability to State Reachability. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II (Lecture Notes in Computer Science)*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.), Vol. 9135. Springer, 95–107. https://doi.org/10.1007/978-3-662-47666-6_8

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2015b. Tractable Refinement Checking for Concurrent Objects. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 651–662. https://doi.org/10.1145/2676726.2677002

Eric A. Brewer. 2000. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA.*, Gil Neiger (Ed.). ACM, 7. https://doi.org/10.1145/343477.343502

Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: a complete and automatic linearizability checker. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 330–340. https://doi.org/10.1145/1806596.1806634

Jacob Burnim, George C. Necula, and Koushik Sen. 2011. Specifying and checking semantic atomicity for multithreaded programs. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, Rajiv Gupta and Todd C. Mowry (Eds.). ACM, 79–90. https://doi.org/10.1145/1950365.1950377

Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert Tappan Morris, and Eddie Kohler. 2015. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. *ACM Trans. Comput. Syst.* 32, 4 (2015), 10:1–10:47. https://doi.org/10.1145/2699681

Mike Dodds, Andreas Haas, and Christoph M. Kirsch. 2015. A Scalable, Correct Time-Stamped Stack. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 233–246. https://doi.org/10.1145/2676726.2676963

William F. Dowling and Jean H. Gallier. 1984. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *J. Log. Program.* 1, 3 (1984), 267–284. https://doi.org/10.1016/0743-1066(84)90014-1

Cezara Dragoi, Ashutosh Gupta, and Thomas A. Henzinger. 2013. Automatic Linearizability Proofs of Concurrent Objects with Cooperating Updates. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, 174–190. https://doi.org/10.1007/978-3-642-39799-8_11

Michael Emmi and Constantin Enea. 2016. Symbolic abstract data type inference. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 513–525. https://doi.org/10.1145/2837614.2837645

Michael Emmi and Constantin Enea. 2017. Exposing Non-Atomic Methods of Concurrent Objects. *CoRR* abs/1706.09305 (2017). arXiv:1706.09305 http://arxiv.org/abs/1706.09305

Michael Emmi, Constantin Enea, and Jad Hamza. 2015. Monitoring refinement via symbolic reasoning. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 260–269. https://doi.org/10.1145/2737924.2737983

Phillip B. Gibbons and Ephraim Korach. 1997. Testing Shared Memories. *SIAM J. Comput.* 26, 4 (1997), 1208–1244. https://doi.org/10.1137/S0097539794279614

Jad Hamza. 2015. On the Complexity of Linearizability. In *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers (Lecture Notes in Computer Science)*, Ahmed Bouajjani and Hugues Fauconnier (Eds.), Vol. 9466. Springer, 308–321. https://doi.org/10.1007/978-3-319-26850-7_21

Klaus Havelund and Grigore Rosu. 2004. Efficient monitoring of safety properties. *STTT* 6, 2 (2004), 158–173. https://doi.org/10.1007/s10009-003-0117-6

Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2013. Aspect-Oriented Linearizability Proofs. In *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings (Lecture Notes in Computer Science)*, Pedro R. D'Argenio and Hernán C. Melgratti (Eds.), Vol. 8052. Springer, 242–256. https://doi.org/10.1007/978-3-642-40184-8_18

Maurice Herlihy and Nir Shavit. 2008. *The art of multiprocessor programming.* Morgan Kaufmann.

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

Alex Horn and Daniel Kroening. 2015. Faster Linearizability Checking via P-Compositionality. In *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings (Lecture Notes in Computer Science)*, Susanne Graf and Mahesh Viswanathan (Eds.), Vol. 9039. Springer, 50–65. https://doi.org/10.1007/978-3-319-19195-9_4

Artem Khyzha, Alexey Gotsman, and Matthew J. Parkinson. 2016. A Generic Logic for Proving Linearizability. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings (Lecture Notes in Computer Science)*, John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.), Vol. 9995. 426–443. https://doi.org/10.1007/978-3-319-48989-6_26

Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 459–470. https://doi.org/10.1145/2462156.2462189

Barbara Liskov and Stephen N. Zilles. 1974. Programming with Abstract Data Types. *SIGPLAN Notices* 9, 4 (1974), 50–59.

Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. 2009. Model Checking Linearizability via Refinement. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings (Lecture Notes in Computer Science)*, Ana Cavalcanti and Dennis Dams (Eds.), Vol. 5850. Springer, 321–337. https://doi.org/10.1007/978-3-642-05089-3_21

Gavin Lowe. 2006. Testing for Linearizability. *Under submission* (2006).

Maged M. Michael. 2004. *ABA Prevention Using Single-Word Instructions.* Technical Report RC 23089. IBM Thomas J. Watson Research Center.

Mark Moir and Nir Shavit. 2004. Concurrent Data Structures. In *Handbook of Data Structures and Applications.*, Dinesh P. Mehta and Sartaj Sahni (Eds.). Chapman and Hall/CRC. https://doi.org/10.1201/9781420035179.ch47

Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Verifying linearizability with hindsight. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, Andréa W. Richa and Rachid Guerraoui (Eds.). ACM, 85–94. https://doi.org/10.1145/1835698.1835722

Gerhard Schellhorn, Heike Wehrheim, and John Derrick. 2012. How to Prove Algorithms Linearisable. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science)*, P. Madhusudan and Sanjit A. Seshia (Eds.), Vol. 7358. Springer, 243–259. https://doi.org/10.1007/978-3-642-31424-7_21

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015a. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 77–87. https://doi.org/10.1145/2737924.2737964

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015b. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer, 333–358. https://doi.org/10.1007/978-3-662-46669-8_14

Ohad Shacham, Nathan Grasso Bronson, Alex Aiken, Mooly Sagiv, Martin T. Vechev, and Eran Yahav. 2011. Testing atomicity of composed concurrent operations. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented*

Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 51–64. https://doi.org/10.1145/2048066.2048073

Viktor Vafeiadis. 2010. Automatically Proving Linearizability. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.), Vol. 6174. Springer, 450–464. https://doi.org/10.1007/978-3-642-14295-6_40

Jeannette M. Wing and C. Gong. 1993. Testing and Verifying Concurrent Objects. *J. Parallel Distrib. Comput.* 17, 1-2 (1993), 164–182. https://doi.org/10.1006/jpdc.1993.1015

Lu Zhang, Arijit Chattopadhyay, and Chao Wang. 2015. Round-Up: Runtime Verification of Quasi Linearizability for Concurrent Data Structures. *IEEE Trans. Software Eng.* 41, 12 (2015), 1202–1216. https://doi.org/10.1109/TSE.2015.2467371

Shao Jie Zhang. 2011. Scalable automatic linearizability checking. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 1185–1187. https://doi.org/10.1145/1985793.1986037