Verifying Concurrent Programs against Sequential Specifications

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza

LIAFA, Université Paris Diderot
{abou,mje,cenea,jhamza}@liafa.univ-paris-diderot.fr

Abstract. We investigate the algorithmic feasibility of checking whether concurrent implementations of shared-memory objects adhere to their given sequential specifications; sequential consistency, linearizability, and conflict serializability are the canonical variations of this problem. While verifying sequential consistency of systems with unbounded concurrency is known to be undecidable, we demonstrate that conflict serializability, and linearizability with fixed linearization points are EXPSPACE-complete, while the general linearizability problem is undecidable.
Our (un)decidability proofs, besides bestowing novel theoretical results, also reveal novel program explorations strategies. For instance, we show that every violation to conflict serializability is captured by a conflict cycle whose length is bounded independently from the number of concurrent operations. This suggests an incomplete detection algorithm which only remembers a small subset of conflict edges, which can be made complete by increasing the number of remembered edges to the cycle-length bound. Similarly, our undecidability proof for linearizability suggests an incomplete detection algorithm which limits the number of "barriers" bisecting non-overlapping operations. Our decidability proof of bounded-barrier linearizability is interesting on its own, as it reduces the consideration of all possible operation serializations to numerical constraint solving. The literature seems to confirm that most violations are detectable by considering very few conflict edges or barriers.

1 Introduction

A key class of correctness criteria for concurrent systems is adherence to better established sequential specifications. Such criteria demand that each concurrent execution of operations corresponds, at the level of abstraction described by the operations' specification, to some serial sequence of the same operations permitted by the specification. For instance, given a conventional specification of a mathematical set, a concurrent execution in which the operations $\mathrm{add}(a), \mathrm{remove}(b), \mathrm{is\_empty}(\mathrm{true}), \mathrm{remove}(a), \mathrm{add}(b)$ overlap could be permitted, though one with only the operations $\mathrm{add}(a)$ and $\mathrm{remove}(b)$ could not.

Variations on this theme of criteria are the accepted correctness conditions for various types of concurrent systems. In the context of processor memory architectures, *sequential consistency (SC)* [24] allows only executions of memory access

[0]The proofs to many of our technical results appear in an extended report [7].

operations for which the same operations taken serially adhere to the specification of individual memory registers—i.e., where each load reads the last-written value. Additionally, any two operations of the serialization carried out by the same process must occur in the same order as in the original concurrent execution. In the context of concurrent data structure implementations, *linearizability* [21] demands additionally that two operations which do not overlap in the original concurrent execution occur in the same order in any valid serialization.

The same kinds of criteria are also used in settings where operation specifications are less abstract. For transactional systems (e.g., databases, and runtime systems which provide atomic sections in concurrent programs), *(strict) serializability* [28] allows only executions for which the same transactions taken serially adhere to the specification of an entire (random-access) memory observable by the transactions; additionally, transactions executed by the same process (or which did not overlap, in the *strict* case) are obliged to occur in the same order in any valid serialization. Practical considerations, such as the complexity of determining whether a given trace is serializable, have generated even more restrictive notions of serializability. *Conflict serializability* (Papadimitriou [28] calls this property "DSR") demands additionally—viewing a serialization as a reordering of actions which untangles the operations of a concurrent execution—that no two *conflicting* actions are reordered in the serialization. The typical definition of "conflict" relates accesses to the same memory location or region, with at least one being a store.

In this work we investigate the fundamental questions about the algorithmic feasibility of verifying concurrent programs with respect to sequential specifications. While our results consider programs with unbounded concurrency arising from, e.g., dynamic thread-creation, they, as do most other (un)decidability results concerning concurrent program analysis, apply to programs where the domain of data values is either finite, or reduced by a finitary abstraction.

While the problem of determining whether a given concurrent system is sequentially consistent with respect to a given sequential specification is known to be undecidable, even when the number of concurrent processes is bounded [1], the decidability of the analogous questions for (conflict) serializability and linearizability, for unbounded systems of concurrent processes, remains open. (Alur et al. [1] have proved both of these problems decidable[1], resp., in PSPACE and EXPSPACE, when the number of concurrent processes is bounded.) In this work we establish these decidability and complexity results for unbounded systems, and as byproduct, uncover program exploration strategies which prioritize the discovery of naturally-occurring property violations.

Our first result, of Section 3, is that conflict serializability is decidable, and complete for exponential space. Since existing techniques rely on cycle detection in an exhaustive exploration of possible conflict relations (graphs) among concurrent operations [17], allowing for an unbounded number of concurrent operations renders these techniques inapplicable to verification, since the unbounded set of possible conflict graphs cannot generally be enumerated in finite time. Contrarily,

[1] The correct decidability proof for serializability is due to Farzan and Madhusudan [17].

here we demonstrate that every cyclic conflict graph contains a cycle which is bounded independently of the number of concurrent operations; this cycle length is instead bounded as a linear function in the number of memory locations. This suggests that an incomplete cycle detection algorithm which only remembers a small subset of conflict edges can be made complete by increasing the number of remembered edges to the given cycle-length bound. Even so, we expect that most violations to conflict serializability can be efficiently detected by remembering very few conflict edges: those we have seen reported in the literature are expressed with length 2 cycles [13, 19], and for systems satisfying certain supposedly-common symmetry conditions, any violation *must* occur with only two threads [19].

Our second result, of Section 4, is that the *static linearizability problem*, in which the so-called "linearization points" of operations which modify the shared-object state are fixed to particular implementation actions, is also decidable, and complete for exponential space. Informally, a *linearization point* of an operation in an execution is a point in time where the operation is conceptually effectuated; given the linearization points of each operation, the only valid serialization is the one which takes operations in order of their linearization points. Although static linearizability is a stronger criterion than linearizability, it is based on a fairly-well established proof technique [21] which is sufficiently weak to prove linearizability of many common concurrent data-structure algorithms [31].

Turning to the general problem, in Section 5, we show that verifying linearizability for unbounded concurrent systems is undecidable. Our proof is a reduction from a reachability problem on counter machines, and relies on imposing an unbounded number of "barriers" which bisect non-overlapping operations in order to encode an unbounded number of zero-tests of the machines' counters. Informally, a barrier is a temporal separation between two non-overlapping operations, across which valid serializations are forbidden from commuting those operations.

Besides disarming our proof of undecidability, bounding the amount of barriers reveals an incomplete algorithm for detecting linearizability violations, by exploring only those expressed with few barriers. Similarly to the small-cycle case in conflict serializability, we expect that most violations to linearizability are detectable with very few barriers; indeed the naturally-occurring bugs we are aware of, including the infamous "ABA" bug [26], induce violations with zero or one barrier. Our decidability proof of bounded-barrier linearizability in Section 6 is interesting on its own, since it effectively reduces the problem of considering all possible serializations of an unbounded number of operations to a numerical constraint solving problem. Using a simple prototype implementation leveraging SMT-based program exploration, we use this reduction to quickly discover bugs known in or injected into textbook concurrent algorithms.

To summarize, the contributions of this work are the first known (un)decidability results for (§3) conflict serializability, (§4) static linearizability, (§5) linearizability, and (§6) bounded-barrier linearizability, for systems with unbounded concurrency. Furthermore, besides substantiating these theoretical results our proofs reveal novel prioritized exploration strategies, based on cycle- and barrier-bounding. Since most known linearizable systems are also static-linearizable, combining static-linearizability with bounded-barrier exploration ought to provide a promising approach for proving either correctness or violation for many practically-occurring systems.

2 Preliminaries

In this work we consider a program model in which an unbounded number of *operations* concurrently access finite-domain shared data. Operations correspond to invocations of a finite *library* of *methods*. Here, methods correspond to the implementations of application programming interface (API) entries of concurrent or distributed data structures, and less conventionally, to the atomic code sections of concurrent programs, or to the SQL implementations of database transactions. A library is then simply the collection of API implementations, or transactional code. Usually concurrent data structure libraries and transactional runtime systems are expected to ensure that executed operations are logically equivalent to some understood serial behavior, regardless of how *clients* concurrently invoke their methods or transactions; the implication is that such systems should function correctly for a *most-general client* which concurrently invokes an unbounded number of methods with arbitrary timing. In what follows we formalize these notions as a basis for formulating our results.

2.1 Unbounded Concurrent Systems

A *method* is a finite automaton $M = \langle Q, \Sigma, I, F, \hookrightarrow \rangle$ with labeled transitions $\langle m_1, v_1 \rangle \xhookrightarrow{a} \langle m_2, v_2 \rangle$ between method-local states $m_1, m_2 \in Q$ paired with finite-domain shared-state valuations $v_1, v_2 \in V$. The initial and final states $I, F \subseteq Q$ represent the method-local states passed to, and returned from, M. A *library* L is a finite set of methods, and we refer to the components of a particular method (resp., library) by subscripting, e.g., the states and symbols Q_M and Σ_M (resp., Q_L and Σ_L). Though here we suppose an abstract notion of shared-state valuations, in later sections we interpret them as valuations to a finite set of finite-domain variables.

A *client* of a library L is a finite automaton $C = \langle Q, \Sigma, \ell_0, \hookrightarrow \rangle$ with initial state $\ell_0 \in Q$ and transitions $\hookrightarrow \subseteq Q \times \Sigma \times Q$ labeled by the alphabet $\Sigma = \{M(m_0, m_f) : M \in L, m_0, m_f \in Q_M\}$ of library method calls; we refer to a client C's components by subscripting, e.g., the states and symbols Q_C and Σ_C. *The most general client* $C^\star = \langle Q, \Sigma, \ell_0, \hookrightarrow \rangle$ of a library L nondeterministically calls L's methods in any order: $Q = \{\ell_0\}$ and $\hookrightarrow \, = Q \times \Sigma \times Q$.

We consider *unbounded concurrent systems* $L[C]$ in which the methods of a library L are invoked by an arbitrary number of concurrent *threads* executing a copy of a given client C; note that any shared memory program with an unbounded number of finite-state threads can be modeled using a suitably-defined client C. A *configuration* $c = \langle v, u \rangle$ of $L[C]$ is a shared memory valuation $v \in V$, along with a map u mapping each thread $t \in \mathbb{N}$ to a tuple $u(t) = \langle \ell, m_0, m \rangle$, composed of a client-local state $\ell \in Q_C$, along with initial and current method

$$\textsc{Internal}\quad \frac{u_1(t) = \langle \ell, m_0, m_1\rangle \qquad \langle m_1, v_1\rangle \xhookrightarrow{a} \langle m_2, v_2\rangle \qquad u_2 = u_1\,(t \mapsto \langle \ell, m_0, m_2\rangle)}{\langle v_1, u_1\rangle \xrightarrow[L[C]]{\langle a,t\rangle} \langle v_2, u_2\rangle}$$

$$\textsc{Call}\quad \frac{u_1(t) = \langle \ell_1, \bot, \bot\rangle \qquad m_0 \in I_M \quad \ell_1 \xhookrightarrow{M(m_0,m_f)}_{C} \ell_2 \qquad u_2 = u_1\,(t \mapsto \langle \ell_1, m_0, m_0\rangle)}{\langle v, u_1\rangle \xrightarrow[L[C]]{\mathsf{call}(M,m_0,t)} \langle v, u_2\rangle}$$

$$\textsc{Return}\quad \frac{u_1(t) = \langle \ell_1, m_0, m_f\rangle \qquad m_f \in F_M \quad \ell_1 \xhookrightarrow{M(m_0,m_f)}_{C} \ell_2 \qquad u_2 = u_1\,(t \mapsto \langle \ell_2, \bot, \bot\rangle)}{\langle v, u_1\rangle \xrightarrow[L[C]]{\mathsf{ret}(M,m_f,t)} \langle v, u_2\rangle}$$

Fig. 1. The transition relation $\rightarrow_{L[C]}$ for the library-client composition $L[C]$.

states $m_0, m \in Q_L \cup \{\bot\}$; $m_0 = m = \bot$ when thread t is not executing a library method. In this way, configurations describe the states of arbitrarily-many threads executing library methods. The transition relation $\rightarrow_{L[C]}$ of $L[C]$ is listed in Figure 1 as a set of operational steps on configurations. A configuration $\langle v, u\rangle$ of $L[C]$ is called v_0-*initial* for a given $v_0 \in V$ when $v = v_0$ and $u(t) = \langle \ell_0, \bot, \bot\rangle$ for all $t \in \mathbb{N}$, where ℓ_0 is the initial state of client C. An *execution* of $L[C]$ is a sequence $\rho = c_0 c_1 \ldots$ of configurations such that $c_i \rightarrow_{L[C]} c_{i+1}$ for all $0 \le i < |\rho|$, and ρ is called v_0-*initial* when c_0 is.

We associate to each concurrent system $L[C]$ a *canonical vector addition systems with states (VASS)*,[2] denoted $\mathcal{A}_{L[C]}$, whose states are the set of shared-memory valuations, and whose vector components count the number of threads in each thread-local state; a transition of $\mathcal{A}_{L[C]}$ from $\langle v_1, \boldsymbol{n}_1\rangle$ to $\langle v_2, \boldsymbol{n}_2\rangle$ updates the shared-memory valuation from v_1 to v_2 and the local state of some thread t from $u_1(t)$ to $u_2(t)$ by decrementing the $u_1(t)$-component of $\boldsymbol{n}_1$, and incrementing the $u_2(t)$-component, to derive $\boldsymbol{n}_2$. Several of our proof arguments in the following sections invoke the canonical VASS simulation of a concurrent system, which we define fully in our extended report [7].

A *call action of thread* t is a symbol $\mathsf{call}(M, m, t)$, a *return action* is a symbol $\mathsf{ret}(M, m, t)$, and an *internal action* is a symbol $\langle a, t\rangle$. We write σ to denote a sequence of actions, and τ to denote a *trace*—i.e., a sequence of actions labeling some execution. An $M[m_0, m_f]$-*operation* θ (or more simply, M-*operation*, or just *operation*) of a sequence σ is a maximal subsequence of actions of some thread t beginning with a call action $\mathsf{call}(M, m_0, t)$, followed by a possibly-empty sequence of internal actions, and possibly ending with a return action $\mathsf{ret}(M, m_f, t)$; $m_f = *$ when θ does not end in a return action. When θ ends with a return action, we say θ is *completed*, and otherwise θ is *pending*; a sequence σ is *complete* when all of its operations are completed. Two operations θ_1 and θ_2 of σ *overlap* when the minimal subsequence of σ containing both θ_1 and θ_2 is neither $\theta_1 \cdot \theta_2$ nor $\theta_2 \cdot \theta_1$. Two non-overlapping operations θ_1 followed by θ_2 in σ are called *serial* when θ_1 is completed; note that all operations of the same thread are serial. A sequence σ is *(quasi) serial* when no two (completed) operations of σ overlap.

A *(strict) permutation* of an action sequence σ containing operations Θ is an action sequence π with operations Θ such that every two same-thread operations

[2]See our extended report [7] for a standard definition of VASS.

(resp., every two serial operations) of σ occur in the same serial order in π. Note that π itself is not necessarily a trace of a system from which σ may be a trace.

2.2 Conflict Serializability

The notion of "conflict serializability" is a restriction to the more liberal "serializability" [28]: besides requiring that each concurrent execution of operations corresponds to some serial sequence, a "conflict relation," relating the individual actions of each operation, must be preserved in deriving that serial sequence from a permutation of actions in the original concurrent execution. Both notions are widely accepted correctness criteria for transactional systems.

We fix a symmetric[3] relation $\prec$ on the internal library actions Σ_L called the *conflict relation*. Although here we assume an abstract notion of conflict, in practice, two actions conflict when both access the same memory location, and at least one affects the value stored in that location; e.g., two writes to the same shared variable would conflict. A permutation π of a trace τ is *conflict-preserving* when every pair $\langle a_1, t_1 \rangle$ and $\langle a_2, t_2 \rangle$ of actions of τ appear in the same order in π whenever $a_1 \prec a_2$. Intuitively, a conflict-preserving permutation w.r.t. the previously-mentioned notion of conflict is equally executable on a sequentially-consistent machine.

Definition 1 (Conflict Serializability [28]). *A trace τ is* conflict serializable *when there exists a conflict-preserving serial permutation of τ.*

This definition extends to executions, to systems $L[C]$ whose executions are all conflict serializable, and to libraries L when C is the most general client $C^\star$.

2.3 Linearizability

Contrary to (conflict) serializability, linearizability [21] is more often used in contexts, such as concurrent data structure libraries, in which an abstract specification of operations' serial behavior is given explicitly. For instance, linearizability with respect to a specification of a concurrent stack implementation would require the abstract push($\cdot$) and pop($\cdot$) operations carried out in a concurrent trace τ correspond to some serial sequence σ of push($\cdot$)s and pop($\cdot$)s, in which each pop(a) can be matched to a previous push(a); Figure 2 illustrates an automaton-based specification of a two-element unary stack. Note that linearizability does not require that a corresponding reordering of the trace τ can actually be executed by this stack implementation, nor that the implementation could have even executed these operations serially.

A *specification* S of a library L is a language over the *specification alphabet*

$$\Sigma_S \stackrel{\text{def}}{=} \{M[m_0, m_f] : M \in L, m_0, m_f \in Q_M\}.$$

In this work we assume specifications are regular languages; in practice, specifications are prefix closed. We refer to the alphabet containing both symbols

[3] All definitions of conflict that we are aware of assume symmetric relations.

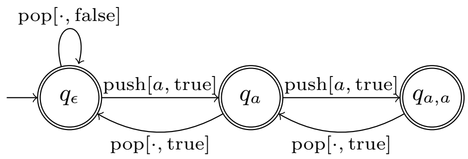


Fig. 2. The sequential specification of two-element stacks containing the (abstract) value a, given as the language of a finite automaton, whose operation alphabet indicates both the argument and return values.

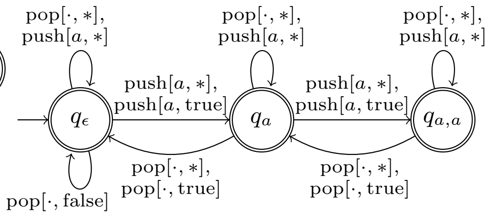


Fig. 3. The pending closure of the stack specification from Figure 2.

$M[m_0, m_f]$ and $M[m_0, *]$ for each $M[m_0, m_f]$ occurring in Σ_S as the *pending-closed* alphabet of S, denoted $\overline{\Sigma}_S$.

Informally, a library L is linearizable w.r.t. a specification S when the operations of any concurrent trace can be serialized to a sequence of operations belonging to S, which must preserve the order between non-overlapping operations. However, the presence of pending operations introduces a subtlety: a trace may be considered linearizable by supposing that certain pending operations have already been effectuated—e.g., a trace of a concurrent stack implementation in which push(a) is pending and pop(a) has successfully completed *is* linearizable—while simultaneously supposing that other pending operations are ignored—e.g., a trace in which push(a) is pending and pop(a) returned false is also linearizable. To account for the possible effects of pending operations, we define a *completion* of a (quasi) serial sequence $\sigma = \theta_1\theta_2 \ldots \theta_i$ of operations to be any sequence $f(\sigma) = f(1)f(2)\ldots f(i)$ for some function f preserving completed operations (i.e., $f(j) = \theta_j$ when θ_j is completed), and either deleting (i.e., $f(j) = \varepsilon$) or completing (i.e., $f(j) = \theta_j \cdot \mathsf{ret}(M, m_f, t)$, for some $m_f \in Q_M$) each $M[m_0, *]$ operation of some thread t. Note that a completion of a (quasi) serial sequence σ is a complete serial sequence. Finally, the *S-image* of a serial sequence σ, denoted $\sigma \mid S$, maps each $M[m_0, m_f]$-operation θ to the symbol $M[m_0, m_f] \in \overline{\Sigma}_S$.

Definition 2 (Linearizability [21]). *A trace τ is* S-linearizable *when there exists a completion*[4] *π of a strict, quasi-serial permutation of τ such that $(\pi|S) \in S$.*

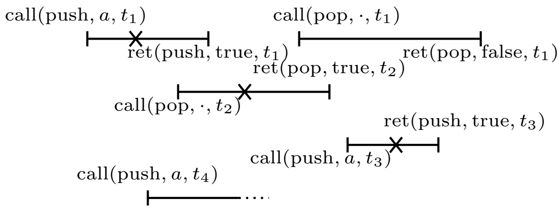
This notion extends naturally to executions of a system $L[C]$, to the system $L[C]$ itself, and to L when C is the most general client $C^\star$.

Example 1. The trace pictured in Figure 4 can be strictly permuted into a quasi-serial sequence whose completion (shown) excludes the pending push operation, and whose S-image

$$\text{push}[a, \text{true}]\ \text{pop}[\cdot, \text{true}]\ \text{pop}[\cdot, \text{false}]\ \text{push}[a, \text{true}]$$

belongs to the stack specification from Figure 2.

[4] Some works give an alternative yet equivalent definition using the completion of a strict, quasi-serial permutation of the S-image, rather than the S-image of a completion.



$$\begin{aligned}&\mathsf{call}(\text{push}, a, t_1) \cdot \mathsf{ret}(\text{push}, \text{true}, t_1)\\ &\cdot \mathsf{call}(\text{pop}, \cdot, t_2) \cdot \mathsf{ret}(\text{pop}, \text{true}, t_2)\\ &\cdot \mathsf{call}(\text{pop}, \cdot, t_1) \cdot \mathsf{ret}(\text{pop}, \text{false}, t_1)\\ &\cdot \mathsf{call}(\text{push}, a, t_3) \cdot \mathsf{ret}(\text{push}, \text{true}, t_3)\end{aligned}$$

Fig. 4. The visualization of a trace τ with four threads executing four completed and one pending operation, along with a completion of a strict, quasi-serial permutation of τ (ignoring internal actions).

2.4 Linearizability with Pending-Closed Specifications

In fact, even though the subtlety arising from pending operations is a necessary complication to the definition of linearizability, for the specifications we consider in this work given by regular languages, this complication can be "compiled away" into the specification itself. This leads to an equivalent notion of linearizability without the need to find a completion of a given quasi-serial operation sequence.

The *pending closure* of a specification S, denoted $\overline{S}$ is the set of S-images of serial sequences which have completions whose S-images are in S:

$$\overline{S} \stackrel{\text{def}}{=} \{(\sigma \mid S) \in \overline{\Sigma}_S^* : \exists \sigma' \in \Sigma_S^*.\ (\sigma' \mid S) \in S \text{ and } \sigma' \text{ is a completion of } \sigma\}.$$

The language of the automaton of Figure 3 is the pending closure of the specification from Figure 2; looping transitions labeled from $\overline{\Sigma}_S \setminus \Sigma_S$ correspond to deleting a pending operation in the completion, while non-loop transitions labeled from $\overline{\Sigma}_S \setminus \Sigma_S$ correspond to completing a pending operation.

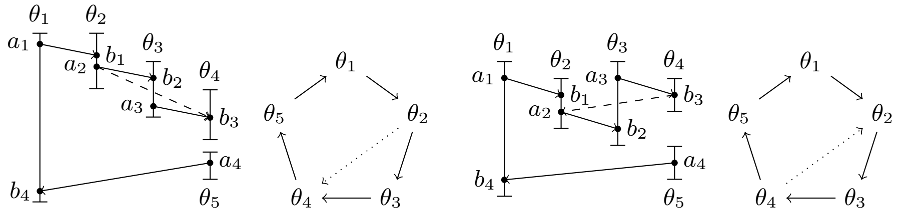
The following straightforward results allow us to suppose that the complication of closing serializations of each trace is compiled away, into the specification.

Lemma 1. *The pending closure $\overline{S}$ of a regular specification S is regular.*

Lemma 2. *A trace τ is S-linearizable if and only if there exists a strict, quasi-serial permutation π of τ such that $(\pi \mid S) \in \overline{S}$.*

3 Deciding Conflict Serializability

Existing procedures for deciding conflict serializability (e.g., of individual traces, or finite-state systems) essentially monitor executions using a "conflict graph" which tracks the conflict relation between concurrent operations; an execution remains conflict serializable as long as the conflict graph remains acyclic, while a cyclic graph indicates a violation to conflict serializability. While the conflict graph can be maintained in polynomial-space when the number of concurrent threads is bounded [17], this graph becomes unbounded as soon as the number of threads does. In this section we demonstrate that there exists an alternative structure witnessing non-conflict-serializability, whose size remains bounded



(a) (b) (c) (d)

Fig. 5. Conflict-violation witness embeddings and their corresponding conflict graph cycles over five operations $\theta_1, \theta_2, \theta_3, \theta_4, \theta_5$. (a) The witness $\langle a_1, b_1\rangle\, \langle a_2, b_2\rangle\, \langle a_3, b_3\rangle\, \langle a_4, b_4\rangle$ is not minimal when $b_2 = b_3$, since $\langle a_1, b_1\rangle\, \langle a_2, b_3\rangle\, \langle a_4, b_4\rangle$ is also a witness. (c) The witness $\langle a_1, b_1\rangle\, \langle a_2, b_2\rangle\, \langle a_3, b_3\rangle\, \langle a_4, b_4\rangle$ is not minimal when $b_2 = b_3$, since $\langle b_3, a_2\rangle\, \langle a_2, b_2\rangle\, \langle a_3, b_3\rangle$ is also a witness. The conflict graphs of (a) and (c) are shown in (b) and (d).

independently of the number of concurrent threads, and which we use to prove EXPSPACE-completeness of conflict-serializability.

Definition 3 (Conflict-Graph [28]). *The* conflict graph *of a trace τ is the directed graph $G_\tau = \langle \Theta, E\rangle$ whose nodes Θ are the operations of τ, and which contains an edge from θ_1 to θ_2 when either:*

- *θ_1 and θ_2 are serial and θ_1 occurs before θ_2 in τ, or*
- *there exist a* conflicting pair *of actions a_1 and a_2 of θ_1 and θ_2, resp., such that $a_1 \prec a_2$ and a_1 occurs before a_2 in τ.*

Although a trace is serializable if and only if its conflict graph is acyclic [17], the size of the conflict graph grows with the number of concurrent operations.

An *embedding* of a sequence of conflicting action pairs $\langle a_1, b_1\rangle \ldots \langle a_k, b_k\rangle$, into a trace τ, is a function f from $\{a_i, b_i : 1 \leq i \leq k\}$ to the actions of τ, such that:

- each $f(a_i)$ is executed by a different thread,
- $f(b_i)$ and $f(a_{\eta(i)})$ are actions of the same thread,
- $f(a_i)$ precedes $f(b_i)$ in τ, and
- $f(b_i)$ precedes $f(a_{\eta(i)})$ in τ when $f(b_i)$ and $f(a_{\eta(i)})$ are of different operations,

for each $1 \leq i \leq k$, where $\eta(i) = (i \bmod k) + 1$. A *conflict-violation witness* for a trace τ is a sequence w for which there exists an embedding into τ.

Example 2. Figure 5a pictures the embeddings of two conflict-violation witnesses containing 4 action pairs, corresponding to a cycle $\theta_1\theta_2\theta_3\theta_4\theta_5\theta_1$ in the conflict graph of Figure 5c associated to the same trace.

The key to decidability of conflict-serializability is that any conflict cycle constructed from two occurrences of the same conflicting action $a \in \Sigma_L$ can be short-circuited into a smaller conflict cycle.

Lemma 3. *A trace τ of a library L (w.r.t. some client C) is not conflict serializable iff there exists a conflict-violation witness for τ of size at most $|\Sigma_L| + 1$.*

Proof. As a direct consequence of our definition, τ is not conflict serializable iff there exists a witness w embedded into τ by some f. (Each w embedded in τ defines a conflict graph cycle, and vice-versa). We show that if some b_i besides b_1 repeats in w, then there exists an even smaller witness w'.

For any $i, j \in \mathbb{N}$ such that $1 < i < j \leq |w|$ and $b_i = b_j$, we consider the two possibilities:

- Suppose $f(b_j)$ occurs after $f(a_i)$ in τ. Then there exists a smaller conflict-violation witness for τ:

$$w' = \langle a_1, b_1 \rangle \dots \langle a_i, b_i \rangle \langle a_{j+1}, b_{j+1} \rangle \dots \langle a_k, b_k \rangle .$$

  The illustration of Figure 5a exemplifies this case when $b_2 = b_3$.
- Suppose $f(b_j)$ occurs before $f(a_i)$ in τ. Then, leveraging the fact that $\prec$ is symmetric, there exists a smaller conflict-violation witness for τ:

$$w' = \langle b_j, a_i \rangle \langle a_i, b_i \rangle \dots \langle a_j, b_j \rangle .$$

  The illustration of Figure 5b exemplifies this case when $b_2 = b_3$.

In either case w is not minimal unless $|w| \leq |\Sigma_L| + 1$. □

As we have considered an abstraction notion of actions which constitute a finite set Σ_L, Lemma 3 would hold equally well for libraries accessing an unbounded shared memory, given an equivalence relation whose quotient set is finite—e.g., by partitioning memory into a finite number of regions—which is obtained in practice by abstraction.

As soon as conflict cycles are bounded, the set of all possible cycles is finitely enumerable. We use this fact to prove that conflict serializability is decidable in exponential space by reduction to state-reachability in VASS, using an extension to the canonical VASS $\mathcal{A}_{L[C]}$ of a given system $L[C]$ (see Section 2.1). We augment the states of $\mathcal{A}_{L[C]}$ to store a (bounded) conflict violation witness w, which is chosen nondeterministically, and incrementally validated as $\mathcal{A}_{L[C]}$ simulates the behavior of $L[C]$. This algorithm is asymptotically optimal, since state-reachability in VASS is also polynomial-time reducible to checking conflict serializability. Our full proof is listed in an extended report [7].

Theorem 1. *The conflict serializability problem for unbounded concurrent systems is EXPSPACE-complete.*

Although exploring all possible conflict cycles up to the bound $|\Sigma_L| + 1$ yields a complete procedure for deciding conflict serializability, we believe that in practice incomplete methods—e.g., based on constraint solving—using much smaller bounds could be more productive. The existing literature on verification of conflict serializability seems to confirm that violations are witnessed with very small cycles; for instance, two different violations on variations to the Transactional Locking II transactional memory algorithm reported by Guerraoui et al. [19] and Dragojević et al. [13] are witnessed by cycles formed by just two pairs of conflicting actions between two operations. Furthermore, Guerraoui et al. [19] show that any violation to conflict serializability in practically-occurring transactional memory systems must occur in an execution with only two threads.

4 Deciding Static Linearizability

Due to the intricacy of checking whether a system is linearizable according to the general notion, of Definition 2, Herlihy and Wing [21] have introduced a stricter criterion, where the so-called "linearization points"—i.e., the points at which operations' effects become instantaneously visible—are specified manually. Though it is sometimes possible to map linearization points to atomic actions in method implementations, generally speaking, the placement of an operation's linearization point can be quite complicated: it may depend on other concurrently executing operations, and it may even reside outside of the operation's execution. Vafeiadis [31] observed that in practice such complicated linearization points arise mainly for "read-only" operations, which do not modify a library's abstract state; a typical example being the contains-operation of an optimistic set [27], whose linearization point may reside in a concurrently executing add- or remove-operation when the contains-operation returns, resp., true or false.

In this section we demonstrate that the *static linearizability* problem, in which the linearization points of non-read-only operations can be statically fixed to implementation actions, is decidable, and complete for exponential space.

Given a method M of a library L and $m_0, m_f \in Q_M$, an $M[m_0, m_f]$-operation θ is *read-only* for a specification S if and only if for all $w_1, w_2, w_3 \in \Sigma_S^*$,

1. If $w_1 \cdot M[m_0, m_f] \cdot w_2 \in S$ then $w_1 \cdot M[m_0, m_f]^k \cdot w_2 \in S$ for all $k \geq 0$, and
2. If $w_1 \cdot M[m_0, m_f] \cdot w_2 \in S$ and $w_1 \cdot w_3 \in S$ then $w_1 \cdot M[m_0, m_f] \cdot w_3 \in S$.

The first condition is a sort of idempotence of $M[m_0, m_f]$ w.r.t. S, while the second says that $M[m_0, m_f]$ does not disable other operations.

Remark 1. Whether an operation is read-only can be derived from the specification. Roughly, an operation $M[m_0, m_f]$ is read-only for a specification given by a finite automaton $\mathcal{A}$ if every transition of $\mathcal{A}$ labeled by $M[m_0, m_f]$ is a self-loop. For instance, the specification in Fig. 2 dictates that pop[$\cdot$, false] is read-only.

The *control graph* $G_M = \langle Q_M, E \rangle$ is the quotient of a method M's transition system by shared-state valuations V: $\langle m_1, a, m_2 \rangle \in E$ iff $\langle m_1, v_1 \rangle \hookrightarrow_{\mathrm{M}}^{\mathrm{a}} \langle m_2, v_2 \rangle$ for some $v_1, v_2 \in V$. A function $\mathsf{LP} : L \to \wp(\Sigma_L)$ is called a *linearization-point mapping* when for each $M \in L$:

1. each symbol $a \in \mathsf{LP}(M)$ labels at most one transition of M,
2. any directed path in G_M contains at most one symbol of $\mathsf{LP}(M)$, and
3. all directed paths in G_M containing $a \in \mathsf{LP}(M)$ reach the same $m_a \in F_M$.

An action $\langle a, i \rangle$ of an M-operation is called a *linearization point* when $a \in \mathsf{LP}(M)$, and operations containing linearization points are said to be *effectuated*; $\mathsf{LP}(\theta)$ denotes the unique linearization point of an effectuated operation θ. A *read-points mapping* $\mathsf{RP} : \Theta \to \mathbb{N}$ for an action sequence σ with operations Θ maps each read-only operation θ to the index $\mathsf{RP}(\theta)$ of an internal θ-action in σ.

Remark 2. One could also define linearization points which depend on predicates involving, e.g., shared-state valuations, loop iteration counts, and return values.

An action sequence σ is called *effectuated* when every completed operation of σ is either effectuated or read-only, and an effectuated completion σ' of σ is *effect preserving* when each effectuated operation of σ also appears in σ'. Given a linearization-point mapping LP, and a read-points mapping RP of an action sequence σ, we say a permutation π of σ is *point preserving* when every two operations of π are ordered by their linearization/read points in σ.

Definition 4. *A trace τ is* $\langle S, \mathsf{LP}\rangle$-linearizable *when τ is effectuated, and there exists a read-points mapping RP of τ, along with an effect-preserving completion π of a strict, point-preserving, and serial permutation of τ such that $(\pi \mid S) \in S$.*

This notion extends naturally to executions of a system $L[C]$, to the system $L[C]$ itself, and to L when C is the most general client $C^\star$.

Definition 5 (Static Linearizability). *The system $L[C]$ is S-*static linearizable *when $L[C]$ is $\langle S, \mathsf{LP}\rangle$-linearizable for some mapping LP.*

Example 3. The execution of Example 1 is $\langle S, \mathsf{LP}\rangle$-linearizable with an LP which assigns points denoted by $\times$s in Figure 4; the completion of a strict, point-preserving, and serial permutation which witnesses this fact is also shown.

Lemma 4. *Every S-static linearizable library is S-linearizable.*

To decide $\langle \mathsf{LP}, S\rangle$-static-linearizability we reduce to a reachability problem on an extension of the given system $L[C]$. The extension simulates the specification automaton $\mathcal{A}_S$, updating its state when operations are effectuated—i.e., at linearization points. Besides ensuring that the method corresponding to each read-only operation θ is enabled in $\mathcal{A}_S$ at some point during θ's execution, our reachability query ensures that each effectuated operation corresponds to an enabled transition in $\mathcal{A}_S$; otherwise the current execution is not S-linearizable, w.r.t. the mapping LP. Technically, we discharge this reachability query via state-reachability on the canonical VASS of $L[C]$'s extension (see Section 2.1), which yields an exponential-space procedure. As the set of possible linearization-point mappings is finite, this procedure is hoisted to an exponential-space procedure for static-linearizability, leveraging Savitch's Theorem. Our proof in our extended report [7] also demonstrates asymptotic optimality, since VASS state-reachability is also polynomial-time reducible to static linearizability.

Theorem 2. *The static linearizability problem for unbounded concurrent systems with regular specifications is EXPSPACE-complete.*

5 Undecidability of Linearizability in the General Case

Though verifying linearizability is decidable for finite-state systems [1], allowing for an unbounded number of concurrent operations lends the power, e.g., to encode unbounded counters. In this section we demonstrate how to harness this power via a reduction from the undecidable state-reachability problem of counter machines to linearizability of unbounded concurrent systems. Technically, given a counter machine $\mathcal{A}$, we construct a library $L_{\mathcal{A}}$ and a specification $S_{\mathcal{A}}$ such that $L_{\mathcal{A}}[C^\star]$ is *not* $S_{\mathcal{A}}$-linearizable exactly when $\mathcal{A}$ has an execution reaching the given target state. In what follows we outline our simulation of $\mathcal{A}$, ignoring several details in order to highlight the crux of our reduction. Our full proof is listed in an extended report [7].

In our simulation of $\mathcal{A}$ the most general client $C^\star$ invokes an arbitrary sequence of methods from the library $L_{\mathcal{A}}$ containing a *transition method* $\mathrm{T}[t]$ for each transition t of $\mathcal{A}$, and an *increment method* $\mathrm{I}[c_i]$, a *decrement method* $\mathrm{D}[c_i]$, and a *zero-test method* $\mathrm{Z}[c_i]$, for each counter c_i of $\mathcal{A}$. As our simulation should allow only concurrent traces which correspond to executions of $\mathcal{A}$, and $C^\star$ is a priori free to invoke operations at arbitrary times, we are faced with constructing the library $L_{\mathcal{A}}$ and specification $S_{\mathcal{A}}$ so that only certain well-formed concurrent traces are permitted. Our strategy is essentially to build $L_{\mathcal{A}}$ to allow only those traces corresponding to valid sequences of $\mathcal{A}$-transitions, and to build $S_{\mathcal{A}}$ to allow only those traces, which either do not reach the target state of $\mathcal{A}$, or which erroneously pass some zero-test—i.e., on a counter whose value is non-zero.

Figure 6 depicts the structure of our simulation, on an $\mathcal{A}$-execution where two increments are followed by two decrements and a zero test, all on the same counter c_1. Essentially we simulate each execution by a trace in which:

1. A sequence $t_1 t_2 \dots t_i$ of $\mathcal{A}$-transitions is modeled by a pairwise-overlapping sequence of $\mathrm{T}[t_1] \cdot \mathrm{T}[t_2] \cdots \mathrm{T}[t_i]$ operations.
2. Each $\mathrm{T}[t]$-operation has a corresponding $\mathrm{I}[c_i]$, $\mathrm{D}[c_i]$, or $\mathrm{Z}[ci]$ operation, depending on whether t is, resp., an increment, decrement, or zero-test transition with counter c_i.
3. Each $\mathrm{I}[c_i]$ operation has a corresponding $\mathrm{D}[c_i]$ operation.
4. For each counter c_i, all $\mathrm{I}[c_i]$ and $\mathrm{D}[c_i]$ between $\mathrm{Z}[c_i]$ operations overlap.
5. For each counter c_i, no $\mathrm{I}[c_i]$ nor $\mathrm{D}[c_i]$ operations overlap with a $\mathrm{Z}[c_i]$ operation.
6. The number of $\mathrm{I}[c_i]$ operations between two $\mathrm{Z}[c_i]$ operations matches the number of $\mathrm{D}[c_i]$ operations.

The library $L_{\mathcal{A}}$ ensures Properties 1–4 using rendezvous synchronization, with six types of signals: a T/T signal between $\mathrm{T}[\cdot]$-operations, and for each counter c_i, T/I, T/D, and T/Z signals between $\mathrm{T}[\cdot]$-operations and, resp., $\mathrm{I}[c_i]$, $\mathrm{D}[c_i]$, and $\mathrm{Z}[c_i]$ operations, an I/D signal between $\mathrm{I}[c_i]$ and $\mathrm{D}[c_i]$ operations, and a T/C signal between $\mathrm{T}[t]$ operations and $\mathrm{I}[c_i]$ or $\mathrm{D}[c_i]$ operations, for zero-testing transitions t. An initial operation (not depicted in Figure 6) initiates a T/T rendezvous with some $\mathrm{T}[t]$ operation. Each $\mathrm{T}[t]$ operation then performs a rendezvous sequence: when t is an increment or decrement of counter c_i, then $\mathrm{T}[t]$ performs a T/T rendezvous, followed by a T/I, resp., T/D for counter c_i, followed by a final T/T rendezvous; when t is a zero-test of counter c_i, $\mathrm{T}[t]$ performs a T/T rendezvous, followed by some arbitrary number of T/Cs for c_i, followed by a T/Z for c_i, and finally a last T/T rendezvous. Each $\mathrm{I}[c_i]$ operation performs T/I, then I/D, and finally T/C rendezvous for counter c_i, while each $\mathrm{D}[c_i]$ operation performs I/D, then T/D, and finally T/C rendezvous for c_i; the $\mathrm{Z}[c_i]$ operations perform a single T/Z rendezvous for c_i. T/T rendezvousing ensures Property 1,

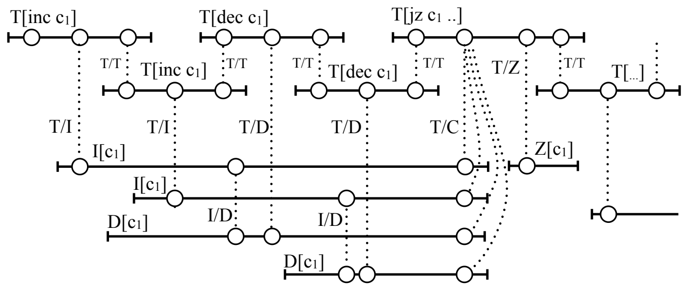


Fig. 6. The $L_{\mathcal{A}}$ simulation of an $\mathcal{A}$-execution with two increments followed by two decrements and a zero-test of counter c_1. Operations are drawn as horizontal lines containing rendezvous actions drawn as circles. Matching rendezvous actions are connected by dotted lines labeled by rendezvous type. Time advances to the right.

T/I, T/D, and T/Z rendezvousing ensures Property 2, I/D rendezvousing ensures Property 3, and T/C rendezvousing ensures Property 4. Note that even in the case where not all pending I[c_i] and D[c_i] operations perform T/C rendezvous with a concurrent T[t] operation, where t is a zero-test transition, at the very least, they overlap with all other pending I[c_i] and D[c_i] operations having performed T/I, resp., T/D, rendezvous since the last Z[c_i] operation.

The trickier part of our proof is indeed ensuring Properties 5 and 6. There we leverage Property 4: when all I[c_i] and D[c_i] operations between two Z[c_i] operations overlap, every permutation of them, including those alternating between I[c_i] and D[c_i] operations, is strict, i.e., is permitted by the definition of linearizability. Our specification $S_{\mathcal{A}}$ takes advantage of this in order to match the unbounded number of I[c_i] and D[c_i] operations using only bounded memory.

Lemma 5. *The specification $S_{\mathcal{A}}$ accepting all sequences which either do not end with a transition to the target state, or in which the number of alternating* I[c_i] *and* D[c_i] *operations between two* Z[c_i] *operations are unequal, is regular.*

Lemma 5 gives a way to ensure Properties 5 and 6, since any trace which is $S_{\mathcal{A}}$-linearizable either does not encode an execution to $\mathcal{A}$'s target state, or respects Property 5 while violating Property 6—i.e., the number of increments and decrements between zero-tests does not match—or violates Property 5: in the latter case, where some I[c_i] or D[c_i] operation θ_1 overlaps with an Z[c_i] operation θ_2, θ_1 can always be commuted over θ_2 to ensure that the number of I[c_i] and D[c_i] operations does not match in some interval between Z[c_i] operations. Thus any trace which is *not* $S_{\mathcal{A}}$-linearizable must respect both Properties 5 and 6. It follows that any trace of $L_{\mathcal{A}}$ which is not $S_{\mathcal{A}}$-linearizable guarantees Properties 1–6, and ultimately corresponds to a valid execution of $\mathcal{A}$, and visa versa, thus reducing counter machine state-reachability to $S_{\mathcal{A}}$-linearizability.

Theorem 3. *The linearizability problem for unbounded concurrent systems with regular specifications is undecidable.*

6 Deciding Bounded Barrier Linearizability

Our proof in Section 5 that verifying linearizability is undecidable relies on constructing an unbounded amount of "barriers" bisecting serial operations in order to encode unboundedly-many zero-tests of a counter machine. Besides disarming our undecidability proof, bounding the number of barriers leads to an interesting heuristic for detecting violations to linearizability, based on the hypothesis that many violations occur in executions expressed with few barriers. In this section we demonstrate not only that the bounded-barrier linearizability problem is decidable, but that when restricting exploration to bounded-barrier executions, checking linearizability reduces to a constraint solving problem on the valuations of counters counting the number of each operation occurring in a finite number of barrier-separated intervals. Similarly to how context-bounding reduces the problem of exploring concurrent program *interleavings* to sequential program behaviors [22], barrier-bounding reduces the problem of exploring concurrent operation *serializations* to counter-constraint solving.

Formally, a *barrier* of a trace τ is an index $0 < B < |\tau|$ such that $\tau(B)$ is a call action, and the nearest preceding non-internal action of τ is a return action. An *interval* is a maximal integer interval $I = [i_1, i_2]$ of τ-indices containing no barriers except i_1, in the case that $i_1 > 0$; we index the intervals of a trace sequentially from 0, as $I_0, I_1, \ldots, I_k$. The *span* of an operation θ of τ is the pair $\langle I_i, I_j \rangle$ of intervals such that θ begins in I_i and ends in I_j—and $I_j = \omega$ when θ is pending. The trace τ of Example 1 contains two barriers, B_1 and B_2, where $\tau(B_1) = \mathsf{call}(\text{pop}, \cdot, t_1)$ and $\tau(B_2) = \mathsf{call}(\text{push}, a, t_3)$, thus dividing τ into three intervals, $I_0 = [0, B_1 - 1]$, $I_1 = [B_1, B_2 - 1]$, and $I_2 = [B_2, |\tau| - 1]$; the span of, e.g., the operation of threads t_2 and t_4 are, resp., $\langle I_0, I_1 \rangle$ and $\langle I_0, \omega \rangle$. Note that the spans of two serial operations of a trace are disjoint.

Definition 6. *The system $L[C]$ is $\langle S, k \rangle$-linearizable when every trace of $L[C]$ with at most k barriers is S-linearizable.*

In what follows we develop the machinery to reduce this bounded-barrier linearizability problem to a reachability problem on systems which count the number of each operation spanning each pair of intervals.

An *interval-annotated alphabet* $\dot{\Sigma} \stackrel{\text{def}}{=} \Sigma \times \mathbb{N} \times (\mathbb{N} \cup \{\omega\})$ attaches (non-zero) interval indices to each symbol of Σ, and an *interval-annotated sequence* $\dot{\sigma} \in \dot{\Sigma}^*$ is *k-bounded* when $i_1 \leq k$ and either $i_2 \leq k$ or $i_2 = \omega$ for each symbol $\langle a, i_1, i_2 \rangle$ of $\dot{\sigma}$. The homomorphism $\dot{h} : \dot{\Sigma} \to \Sigma$ maps each symbol $\langle a, \_, \_ \rangle$ to $\dot{h}(\langle a, \_, \_ \rangle) = a$. An interval-annotated sequence $\dot{\sigma}$ is *timing consistent* when $i_1 \leq i_2$, $i_3 \leq i_4$, and $i_1 \leq i_4$ for any symbol $\langle \_, i_1, i_2 \rangle$ occurring before $\langle \_, i_3, i_4 \rangle$ in $\dot{\sigma}$.

We say that the sequence over the interval-annotated (and pending closed, see Section 2.4) specification alphabet $\dot{\sigma} \in \dot{\Sigma}_S^*$ is *consistent* when $\dot{\sigma}$ is timing consistent, and $i_2 = \omega$ iff $m_f = *$, for all symbols $\langle M[m_0, m_f], i_1, i_2 \rangle$ of $\dot{\sigma}$. The *(k-bounded) interval-annotated specification* $\dot{S}$ of a specification S is the language containing all consistent interval-annotated sequences $\dot{\sigma}$ such that $h(\dot{\sigma}) \in S$. For example, we obtain the 1-bounded interval-annotated specification from the specification of Figure 3 by attaching the interval indices $\langle 1, \omega \rangle$ to each $\mathrm{pop}[\cdot, *]$ and $\mathrm{push}[a, *]$ symbol, and $\langle 1, 1 \rangle$ to each $\mathrm{pop}[\cdot, \mathrm{false}]$, $\mathrm{pop}[\cdot, \mathrm{true}]$, and $\mathrm{push}[a, \mathrm{true}]$ symbol.

Lemma 6. *The k-bounded interval-annotated specification $\dot{S}$, of a regular specification S, is also regular.*

Proof. For any given $k > 0$ the set $W \subseteq \dot{\Sigma}_S^*$ of k-bounded consistent interval-annotated sequences is regular. As regular languages are closed under inverse homomorphism and intersection, $\dot{S} = W \cap \dot{h}^{-1}(S)$ is also regular. □

To relate traces to an interval-annotated specification $\dot{S}$, we define the *interval-annotated S-image* $\dot{\sigma}$ of an action sequence σ as the multiset $\dot{\sigma} : \dot{\Sigma}_S \to \mathbb{N}$ mapping each $\langle M[m_0, m_f], i_1, i_2 \rangle \in \dot{\Sigma}_S$ to the number of occurrences of $M[m_0, m_f]$-operations in σ with span $\langle i_1, i_2 \rangle$.

Example 4. The interval-annotated image $\dot{\tau}$ of the trace τ from Example 1 maps the interval-annotated symbols

$$\mathrm{push}[a, \mathrm{true}][1, 1],\ \mathrm{push}[a, *][1, \omega],\ \mathrm{pop}[\cdot, \mathrm{true}][1, 2],$$
$$\mathrm{pop}[\cdot, \mathrm{false}][2, 3],\ \text{and}\ \mathrm{push}[a, \mathrm{true}][3, 3]$$

to 1, and the remaining symbols of $\dot{\Sigma}_S$ to zero.

Annotating operations with the intervals in which they occur allows a compact representation of specifications' ordering constraints, while abstracting away the order of same-interval operations—as they are free to commute. To realize this abstraction, we recall that the *Parikh image* of a sequence $\sigma \in \Sigma^*$ is the multiset $\Pi(\sigma) : \Sigma \to \mathbb{N}$ mapping each symbol $a \in \Sigma$ to the number of occurrences of a in σ. The *Parikh image* of a language $L \subseteq \Sigma^*$ are the images $\Pi(L) \stackrel{\mathrm{def}}{=} \{\Pi(\sigma) : \sigma \in L\}$ of sequences in L. We prove the following key lemma in our extended report [7].

Lemma 7. *A trace τ with at most k barriers is S-linearizable iff $\dot{\tau} \in \Pi(\dot{S})$, where $\dot{S}$ is the $(k+1)$-bounded interval-annotated specification of S.*

Lemma 7 essentially allows us to reduce the bounded-barrier linearizability problem to a reachability problem: given a trace τ with at most k barriers, τ is linearizable so long as its image $\dot{\tau}$ is included in the Parikh image of the $(k+1)$-bounded specification $\dot{S}$. In effect, rather than considering all possible serializations of τ, it suffices to keep count of the number of pending and completed operations over each span of intervals, and ensure that these counts continually remain within the semi-linear set of counts allowed by the specification. For the purposes of our results here, we keep these counts by increasing the dimension of the canonical vector addition system $\mathcal{A}_{L[C]}$ (see Section 2.1) of a given system $L[C]$. Furthermore, since Bouajjani and Habermehl [6] prove that checking whether reachable VASS configurations lie within a semi-linear set is itself reducible to VASS reachability, and the Parikh image of a regular set is a semi-linear, ensuring these counts continually remain within those allowed by the specification is therefore reducible to VASS reachability. In fact, our proof in our extended report [7] shows this reduction-based procedure is asymptotically optimal, since VASS reachability is also polynomial-time reducible to to $\langle S, k \rangle$-linearizability.

Theorem 4. *The bounded-barrier linearizability problem for unbounded concurrent systems with regular specifications is decidable, and asymptotically equivalent to VASS reachability.*

Theorem 4 holds for any class of specifications with semi-linear Parikh images, including, e.g., context-free languages. Furthermore, though Theorem 4 leverages our reduction from serializations to counting operations for decidability with unbounded concurrent systems, in principle this reduction applies to any class of concurrent systems, including infinite-data systems—without any guarantee of decidability—provided the ability to represent suitable constraints on the counters of annotated specification alphabet symbols. We believe this reduction is valuable whether or not data and/or concurrency are bounded, since we avoid the explicit enumeration of possible serializations.

As a proof of concept, we have implemented a prototype of our reduction. First we instrument a given library implementation (written in Boogie) with (1) auxiliary counters, counting the number of each operation within each bounded span, (2) with Presburger assertions over these counters, encoding the legal specification images, and (3) with a client nondeterministically invoking methods with arbitrary arguments. As a second step we translate this instrumented (concurrent) program to a sequential (Boogie) program, encoding a subset of delay-bounded executions [16], then discover assertion violations using an SMT-based sequential reachability engine [23]. Note that the bounded-barrier reduction, which treats operation serialization, composes naturally with the bounded-delay reduction, which treats operation interleaving. Furthermore, the reduction to SMT allows us to analyze infinite-data implementations; e.g., we analyze an unbounded stack with arbitrary data values, according to a specification which ensures each pop is preceded by a matching push—which is context-free, thus has a semi-linear Parikh image—while ignoring the pushed and popped values.

We have applied our prototype to discover bugs known in or manually-injected into several textbook concurrent data structure algorithms; the resulting linearizability violations are discovered within a few seconds to minutes. Besides evidence to the practical applicability of our reduction algorithm, our small set of experiments suggests that many linearizability violations occur with very few barriers; we discover violations arising from the infamous "ABA" bug [26], along with bugs injected into a 2-lock queue, a lock-coupling set, and Treiber's stack, in executions *without any* barriers. For instance, in an improperly-synchronized Treiber-style stack algorithm, two concurrent pop(a) operations may erroneously remove the same element added by one concurrent push(a) operation; however, no serialization of pop(a), pop(a), and push(a) is included in our stack specification.

Of course, some violations do require barriers. A very simple example is a violation involving one pop(a) serial with one push(a) operation, though since pop(a) and push(a) are not concurrent, a bug causing this violation is unlikely. More interestingly, a lost update due to improper synchronization between two concurrent inc() operations in a zero-initialized counter can only be observed as a linearizability violation when a barrier prevents, e.g., a subsequent read(1) operation from commuting over an inc() operation.

7 Related Work

Papadimitriou [28] and Gibbons and Korach [18] studied variations on the problems of deciding serializability, sequential consistency, and linearizability for single concurrent traces, finding the general problems to be NP-complete, and pointing out several PTIME variants, e.g., when serializations must respect a suitable conflict-order. Alur et al. [1] studied the complexity of similar decision problems for *all* traces of finite-state concurrent systems: while sequential consistency already becomes undecidable for finite-state systems—though Bingham [4] proposes certain decidable pathology-omitting variations—checking conflict serializability is declared PSPACE-complete[5] while linearizability is shown to be in EXPSPACE. Our work considers the complexity of these problems for systems where the number of concurrent operations is unbounded.

Though many have developed techniques for proving linearizability [33, 2, 32, 3, 25, 14, 27, 31, 34, 10], we are not aware of decidability or complexity results for the corresponding linearizability and static linearizability verification problems for unbounded systems. While a few works propose testing-based detection of linearizability violations [9, 11, 10], they rely on explicit enumeration of possible serializations; prioritizing the search for violations with few barriers, and the resulting reduction to numerical constraint solving, are novel.

Several works have also developed techniques for verifying sequential consistency [20, 29, 5, 8] and serializability [12, 30, 17, 19, 15]; Farzan and Madhusudan [17] demonstrate a complete technique for verifying conflict serializability with a bounded number of concurrent operations, and while Guerraoui et al. [19] identify symmetry conditions on transactional systems with which conflict serializability can be verified completely, for an unbounded number of concurrent operations, they propose no means of *checking* that these symmetry conditions hold on any given system. On the contrary, we show that verifying conflict serializability without bounding the number of concurrent operations is EXPSPACE-complete.

References

[1] R. Alur, K. L. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000.

[2] D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV '07: Proc. 19th Intl. Conf. on Computer Aided Verification*, volume 4590 of *LNCS*, pages 477–490. Springer, 2007.

[3] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv. Thread quantification for concurrent shape analysis. In *CAV '08: Proc. 20th Intl. Conf. on Computer Aided Verification*, volume 5123 of *LNCS*, pages 399–413. Springer, 2008.

[4] J. Bingham. *Model Checking Sequential Consistency and Parameterized Protocols.* PhD thesis, The University of British Columbia, August 2005.

[5] The correct proof of PSPACE-completeness is given by Farzan and Madhusudan [17].

[5] J. D. Bingham, A. Condon, A. J. Hu, S. Qadeer, and Z. Zhang. Automatic verification of sequential consistency for unbounded addresses and data values. In *CAV '04: Proc. 16th Intl. Conf. on Computer Aided Verification*, volume 3114 of *LNCS*, pages 427–439. Springer, 2004.

[6] A. Bouajjani and P. Habermehl. Constrained properties, semilinear systems, and petri nets. In *CONCUR '96: Proc. 7th Intl. Conf. on Concurrency Theory*, volume 1119 of *LNCS*, pages 481–497. Springer, 1996.

[7] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Verifying concurrent programs against sequential specifications. Technical report, January 2013.

[8] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *PLDI 07: Proc. ACM SIGPLAN 2007 Conf. on Programming Language Design and Implementation*, pages 12–21. ACM, 2007.

[9] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: a complete and automatic linearizability checker. In *PLDI '10: Proc. 2010 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 330–340. ACM, 2010.

[10] S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In *ESOP '12: Proc. 21st European Symp. on Programming*, volume 7211 of *LNCS*, pages 87–107. Springer, 2012.

[11] J. Burnim, G. C. Necula, and K. Sen. Specifying and checking semantic atomicity for multithreaded programs. In *ASPLOS '11: Proc. 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 79–90. ACM, 2011.

[12] A. Cohen, J. W. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck. Verifying correctness of transactional memories. In *FMCAD 07: Proc. 7th Intl. Conf. on Formal Methods in Computer-Aided Design*, pages 37–44. IEEE Computer Society, 2007.

[13] A. Dragojević, R. Guerraoui, and M. Kapalka. Dividing transactional memories by zero. In *TRANSACT '08: Proc. 3rd ACM SIGPLAN Workshop on Transactional Computing*. ACM, 2008.

[14] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Taşıran. Simplifying linearizability proofs with reduction and abstraction. In *TACAS '10: Proc. 16th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *LNCS*, pages 296–311. Springer, 2010.

[15] M. Emmi, R. Majumdar, and R. Manevich. Parameterized verification of transactional memories. In *PLDI '10: Proc. 2010 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 134–145. ACM, 2010.

[16] M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In *POPL '11: Proc. 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 411–422. ACM, 2011.

[17] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *CAV '08: Proc. 20th Intl. Conf. on Computer Aided Verification*, volume 5123 of *LNCS*, pages 52–65. Springer, 2008.

[18] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM J. Comput.*, 26(4):1208–1244, 1997.

[19] R. Guerraoui, T. A. Henzinger, and V. Singh. Model checking transactional memories. *Distributed Computing*, 22(3):129–145, 2010.

[20] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In *CAV '99: Proc. 11th Intl. Conf. on Computer Aided Verification*, volume 1633 of *LNCS*, pages 301–315. Springer, 1999.

[21] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[22] A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[23] A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *CAV '12: Proc. 24th Intl. Conf. on Computer Aided Verification*, volume 7358 of *LNCS*, pages 427–443. Springer, 2012.

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

[25] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. In *FM '09: Proc. Second World Congress on Formal Methods*, volume 5850 of *LNCS*, pages 321–337. Springer, 2009.

[26] M. M. Michael. ABA prevention using single-word instructions. Technical Report RC 23089, IBM Thomas J. Watson Research Center, January 2004.

[27] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *PODC '10: Proc. 29th Annual Symp. on Principles of Distributed Computing*, pages 85–94. ACM, 2010.

[28] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[29] S. Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. *IEEE Trans. Parallel Distrib. Syst.*, 14(8):730–741, 2003.

[30] S. Taşıran. A compositional method for verifying software transactional memory implementations. Technical Report MSR-TR-2008-56, Microsoft Research, April 2008.

[31] V. Vafeiadis. Automatically proving linearizability. In *CAV '10: Proc. 22nd Intl. Conf. on Computer Aided Verification*, volume 6174 of *LNCS*, pages 450–464. Springer, 2010.

[32] M. T. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI '08: Proc. ACM SIGPLAN 2008 Conf. on Programming Language Design and Implementation*, pages 125–135. ACM, 2008.

[33] L. Wang and S. D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *PPOPP '05: Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 61–71. ACM, 2005.

[34] S. J. Zhang. Scalable automatic linearizability checking. In *ICSE '11: Proc. 33rd Intl. Conf. on Software Engineering*, pages 1185–1187. ACM, 2011.