On Reducing Linearizability to State Reachability[⋆]

Ahmed Bouajjani[1], Michael Emmi[2], Constantin Enea[1], and Jad Hamza[1]

[1] LIAFA, Université Paris Diderot, France
[2] IMDEA Software Institute, Spain

Abstract. Efficient implementations of atomic objects such as concurrent stacks and queues are especially susceptible to programming errors, and necessitate automatic verification. Unfortunately their correctness criteria — linearizability with respect to given ADT specifications — are hard to verify. Even on classes of implementations where the usual temporal safety properties like control-state reachability are decidable, linearizability is undecidable.
In this work we demonstrate that verifying linearizability for certain *fixed* ADT specifications is reducible to control-state reachability, despite being harder for *arbitrary* ADTs. We effectuate this reduction for several of the most popular atomic objects. This reduction yields the first decidability results for verification without bounding the number of concurrent threads. Furthermore, it enables the application of existing safety-verification tools to linearizability verification.

1 Introduction

Efficient implementations of atomic objects such as concurrent queues and stacks are difficult to get right. Their complexity arises from the conflicting design requirements of maximizing efficiency/concurrency with preserving the appearance of atomic behavior. Their correctness is captured by *observational refinement*, which assures that all behaviors of programs using these efficient implementations would also be possible were the atomic reference implementations used instead. Linearizability [12], being an equivalent property [8, 4], is the predominant proof technique: one shows that each concurrent execution has a linearization which is a valid sequential execution according to a specification, given by an abstract data type (ADT) or reference implementation.

Verifying automatically[3] that all executions of a given implementation are linearizable with respect to a given ADT is an undecidable problem [3], even on the typical classes of implementations for which the usual temporal safety properties are decidable, e.g., on finite-shared-memory programs where each thread is a finite-state machine. What makes linearization harder than typical temporal safety properties like control-state reachability is the existential quantification of a valid linearization per execution.

In this work we demonstrate that verifying linearizability for certain *fixed* ADTs is reducible to control-state reachability, despite being harder for *arbitrary* ADTs. We believe that fixing the ADT parameter of the verification problem is justified, since in practice, there are few ADTs for which specialized concurrent implementations

⋆ This work is supported in part by the VECOLIB project (ANR-14-CE28-0018).

[3] Without programmer annotation — see Section 6 for further discussion.

have been developed. We provide a methodology for carrying out this reduction, and instantiate it on four ADTs: the atomic queue, stack, register, and mutex.

Our reduction to control-state reachability holds on any class of implementations which is closed under intersection with regular languages[4] and which is *data independent* — informally, that implementations can perform only read and write operations on the data values passed as method arguments. From the ADT in question, our approach relies on expressing its violations as a finite union of regular languages.

In our methodology, we express the atomic object specifications using inductive rules to facilitate the incremental construction of valid executions. For instance in our atomic queue specification, one rule specifies that a dequeue operation returning empty can be inserted in any execution, so long as each preceding enqueue has a corresponding dequeue, also preceding the inserted empty-dequeue. This form of inductive rule enables a locality to the reasoning of linearizability violations.

Intuitively, first we prove that a sequential execution is invalid if and only if some subsequence could not have been produced by one of the rules. Under certain conditions this result extends to concurrent executions: an execution is not linearizable if and only if some projection of its operations cannot be linearized to a sequence produced by one of the rules. We thus correlate the finite set of inductive rules with a finite set of classes of non-linearizable concurrent executions. We then demonstrate that each of these classes of non-linearizable executions is regular, which characterizes the violations of a given ADT as a finite union of regular languages. The fact that these classes of non-linearizable executions can be encoded as regular languages is somewhat surprising since the number of data values, and thus alphabet symbols, is, a priori, unbounded. Our encoding thus relies on the aforementioned *data independence* property.

To complete the reduction to control-state reachability, we show that linearizability is equivalent to the emptiness of the language intersection between the implementation and finite union of regular violations. When the implementation is a finite-shared-memory program with finite-state threads, this reduces to the coverability problem for Petri nets, which is decidable, and EXPSPACE-complete.

To summarize, our contributions are:

- a generic reduction from linearizability to control-state reachability,
- its application to the atomic queue, stack, register, and mutex ADTs,
- the methodology enabling this reduction, which can be reused on other ADTs, and
- the first decidability results for linearizability without bounding the number of concurrent threads.

Besides yielding novel decidability results, our reduction paves the way for the application of existing safety-verification tools to linearizability verification.

Section 2 outlines basic definitions. Section 3 describes a methodology for inductive definitions of data structure specifications. In Section 4 we identify conditions under which linearizability can be reduced to control-state reachability, and demonstrate that typical atomic objects satisfy these conditions. Finally, we prove decidability of linearizability for finite-shared-memory programs with finite-state threads in Section 5. Proofs to technical results appear in the extended version of this paper [5].

[4] We consider languages of well-formed method call and return actions, e.g., for which each return has a matching call.

2 Linearizability

We fix a (possibly infinite) set $\mathbb{D}$ of *data values*, and a finite set $\mathbb{M}$ of *methods*. We consider that methods have exactly one argument, or one return value. Return values are transformed into argument values for uniformity.[5] In order to differentiate methods taking an argument (e.g., the *Enq* method which inserts a value into a queue) from the other methods, we identify a subset $\mathbb{M}_{in} \subseteq \mathbb{M}$ of *input* methods which do take an argument. A *method event* is composed of a method $m \in \mathbb{M}$ and a data value $x \in \mathbb{D}$, and is denoted $m(x)$. We define the *concatenation* of method-event sequences $u \cdot v$ in the usual way, and ϵ denotes the empty sequence.

Definition 1. *A* sequential execution *is a sequence of method events,*

The projection $u_{|D}$ of a sequential execution u to a subset $D \subseteq \mathbb{D}$ of data values is obtained from u by erasing all method events with a data value not in D. The set of projections of u is denoted $\mathsf{proj}(u)$. We write $u \setminus x$ for the projection $u_{|\mathbb{D}\setminus\{x\}}$.

Example 1. The projection $Enq(1)Enq(2)Deq(1)Enq(3)Deq(2)Deq(3) \setminus 1$ is equal to $Enq(2)Enq(3)Deq(2)Deq(3)$.

We also fix an arbitrary infinite set $\mathbb{O}$ of operation (identifiers). A *call action* is composed of a method $m \in \mathbb{M}$, a data value $x \in \mathbb{D}$, an operation $o \in \mathbb{O}$, and is denoted $\mathtt{call}_o\ m(x)$. Similarly, a *return action* is denoted $\mathtt{ret}_o\ m(x)$. The operation o is used to match return actions to their call actions.

Definition 2. *A* (concurrent) execution *e is a sequence of call and return actions which satisfy a well-formedness property: every return has a call action before it in e, using the same tuple m, x, o, and an operation o can be used only twice in e, once in a call action, and once in a return action.*

Example 2. The sequence $\mathtt{call}_{o_1}\ Enq(7) \cdot \mathtt{call}_{o_2}\ Enq(4) \cdot \mathtt{ret}_{o_1}\ Enq(7) \cdot \mathtt{ret}_{o_2}\ Enq(4)$ is an execution, while $\mathtt{call}_{o_1}\ Enq(7) \cdot \mathtt{call}_{o_2}\ Enq(4) \cdot \mathtt{ret}_{o_1}\ Enq(7) \cdot \mathtt{ret}_{o_1}\ Enq(4)$ and $\mathtt{call}_{o_1}\ Enq(7) \cdot \mathtt{ret}_{o_1}\ Enq(7) \cdot \mathtt{ret}_{o_2}\ Enq(4)$ are not.

Definition 3. *An* implementation $\mathcal{I}$ *is a set of (concurrent) executions.*

Implementations represent libraries whose methods are called by external programs, giving rise to the following closure properties [4]. In the following, c denotes a call action, r denotes a return action, a denotes any action, and e, e' denote executions.

- Programs can call library methods at any point in time:
  $e \cdot e' \in \mathcal{I}$ implies $e \cdot c \cdot e' \in \mathcal{I}$ so long as $e \cdot c \cdot e'$ is well formed.
- Calls can be made earlier:
  $e \cdot a \cdot c \cdot e' \in \mathcal{I}$ implies $e \cdot c \cdot a \cdot e' \in \mathcal{I}$.

[5] Method return values are guessed nondeterministically, and validated at return points. This can be handled using the `assume` statements of typical formal specification languages, which only admit executions satisfying a given predicate. The argument value for methods without argument or return values, or with fixed argument/return values, is ignored.

- Returns been made later:
  $e \cdot r \cdot a \cdot e' \in \mathcal{I}$ implies $e \cdot a \cdot r \cdot e' \in \mathcal{I}$.

Intuitively, these properties hold because call and return actions are not visible to the other threads which are running in parallel.

For the remainder of this work, we consider only *completed* executions, where each call action has a corresponding return action. This simplification is sound when implementation methods can always make progress in isolation [11]: formally, for any execution e with pending operations, there exists an execution e' obtained by extending e only with the return actions of the pending operations of e. Intuitively this means that methods can always return without any help from outside threads, avoiding deadlock.

We simply reasoning on executions by abstracting them into *histories*.

Definition 4. *A* history *is a* labeled partial order $(O, <, l)$ *with* $O \subseteq \mathbb{O}$ *and* $l : O \to \mathbb{M} \times \mathbb{D}$.

The order $<$ is called the *happens-before relation*, and we say that o_1 *happens before* o_2 when $o_1 < o_2$. Since histories arise from executions, their happens-before relations are *interval orders* [4]: for distinct o_1, o_2, o_3, o_4, if $o_1 < o_2$ and $o_3 < o_4$ then either $o_1 < o_4$, or $o_3 < o_2$. Intuitively, this comes from the fact that concurrent threads share a notion of global time. $\mathbb{D}_h \subseteq \mathbb{D}$ denotes the set of data values appearing in h.

The *history of an execution e* is defined as $(O, <, l)$ where:

- O is the set of operations which appear in e,
- $o_1 < o_2$ iff the return action of o_1 is before the call action of o_2 in e,
- an operation o occurring in a call action $\texttt{call}_{\texttt{o}}\ m(x)$ is labeled by $m(x)$.

Example 3. The history of the execution $\texttt{call}_{\texttt{o}_1}\ \mathit{Enq}(7) \cdot \texttt{call}_{\texttt{o}_2}\ \mathit{Enq}(4) \cdot \texttt{ret}_{\texttt{o}_1}\ \mathit{Enq}(7) \cdot \texttt{ret}_{\texttt{o}_2}\ \mathit{Enq}(4)$ is $(\{o_1, o_2\}, <, l)$ with $l(o_1) = \mathit{Enq}(7)$, $l(o_2) = \mathit{Enq}(4)$, and with $<$ being the empty order relation, since o_1 and o_2 *overlap*.

Let $h = (O, <, l)$ be a history and u a sequential execution of length n. We say that h is *linearizable with respect to u*, denoted $h \sqsubseteq u$, if there is a bijection $f : O \to \{1, \dots, n\}$ s.t.

- if $o_1 < o_2$ then $f(o_1) < f(o_2)$,
- the method event at position $f(o)$ in u is $l(o)$.

Definition 5. *A history h is* linearizable *with respect to a set $\mathcal{S}$ of sequential executions, denoted $h \sqsubseteq \mathcal{S}$, if there exists $u \in \mathcal{S}$ such that $h \sqsubseteq u$.*

A set of histories H is *linearizable* with respect to $\mathcal{S}$, denoted $H \sqsubseteq \mathcal{S}$ if $h \sqsubseteq \mathcal{S}$ for all $h \in H$. We extend these definitions to executions according to their histories.

A sequential execution u is said to be *differentiated* if, for all input methods $m \in \mathbb{M}_{in}$, and every $x \in \mathbb{D}$, there is at most one method event $m(x)$ in u. The subset of differentiated sequential executions of a set $\mathcal{S}$ is denoted by $\mathcal{S}_{\neq}$. The definition extends to (sets of) executions and histories. For instance, an execution is differentiated if for all input methods $m \in \mathbb{M}_{in}$ and every $x \in \mathbb{D}$, there is at most one call action $\texttt{call}_{\texttt{o}}\ m(x)$.

Example 4. $\texttt{call}_{\texttt{o}_1}\ \mathit{Enq}(7) \cdot \texttt{call}_{\texttt{o}_2}\ \mathit{Enq}(7) \cdot \texttt{ret}_{\texttt{o}_1}\ \mathit{Enq}(7) \cdot \texttt{ret}_{\texttt{o}_2}\ \mathit{Enq}(7)$ is not differentiated, as there are two call actions with the same input method (Enq) and the same data value.

A *renaming* r is a function from $\mathbb{D}$ to $\mathbb{D}$. Given a sequential execution (resp., execution or history) u, we denote by $r(u)$ the sequential execution (resp., execution or history) obtained from u by replacing every data value x by $r(x)$.

Definition 6. *The set of sequential executions (resp., executions or histories) $\mathcal{S}$ is* data independent *if:*

– *for all $u \in \mathcal{S}$, there exists $u' \in \mathcal{S}_{\neq}$, and a renaming r such that $u = r(u')$,*
– *for all $u \in \mathcal{S}$ and for all renaming r, $r(u) \in \mathcal{S}$.*

When checking that a data-independent implementation $\mathcal{I}$ is linearizable with respect to a data-independent specification $\mathcal{S}$, it is enough to do so for differentiated executions [1]. Thus, in the remainder of the paper, we focus on characterizing linearizability for differentiated executions, rather than arbitrary ones.

Lemma 1 (Abdulla et al. [1]). *A data-independent implementation $\mathcal{I}$ is linearizable with respect to a data-independent specification $\mathcal{S}$, if and only if $\mathcal{I}_{\neq}$ is linearizable with respect to $\mathcal{S}_{\neq}$.*

3 Inductively-Defined Data Structures

A *data structure* $\mathcal{S}$ is given syntactically as an ordered sequence of rules $R_1, \ldots, R_n$, each of the form $u_1 \cdot u_2 \cdots u_k \in \mathcal{S} \wedge Guard(u_1, \ldots, u_k) \Rightarrow Expr(u_1, \ldots, u_k) \in \mathcal{S}$, where the variables u_i are interpreted over method-event sequences, and

– $Guard(u_1, \ldots, u_k)$ is a conjunction of conditions on $u_1, \ldots, u_k$ with atoms
  • $u_i \in M^*$ ($M \subseteq \mathbb{M}$)
  • $\mathsf{matched}(m, u_i)$
– $Expr(u_1, \ldots, u_k)$ is an *expression* $E = a_1 \cdot a_2 \cdots a_l$ where
  • $u_1, \ldots, u_k$ appear in that order, exactly once, in E,
  • each a_i is either some u_j, a method m, or a Kleene closure m^* ($m \in \mathbb{M}$),
  • a method $m \in \mathbb{M}$ appears at most once in E.

We allow k to be 0 for base rules, such as $\epsilon \in \mathcal{S}$.

A condition $u_i \in M^*$ ($M \subseteq \mathbb{M}$) is satisfied when the methods used in u_i are all in M. The predicate $\mathsf{matched}(m, u_i)$ is satisfied when, for every method event $m(x)$ in u_i, there exists another method event in u_i with the same data value x.

Given a sequential execution $u = u_1 \cdot \ldots \cdot u_k$ and an expression $E = Expr(u_1, \ldots, u_k)$, we define $[\![E]\!]$ as the set of sequential executions which can be obtained from E by replacing the methods m by a method event $m(x)$ and the Kleene closures m^* by 0 or more method events $m(x)$. All method events must use the same data value $x \in \mathbb{D}$.

A rule $R \equiv u_1 \cdot u_2 \cdots u_k \in \mathcal{S} \wedge Guard(u_1, \ldots, u_k) \Rightarrow Expr(u_1, \ldots, u_k) \in \mathcal{S}$ is applied to a sequential execution w to obtain a new sequential execution w' from the set:

$$\bigcup_{\substack{w = w_1 \cdot w_2 \cdots w_k \wedge \\ Guard(w_1, \ldots, w_k)}} [\![Expr(w_1, \ldots, w_k)]\!]$$

We denote this $w \xrightarrow{R} w'$. The set of sequential executions $[\![\mathcal{S}]\!] = [\![R_1, \ldots, R_n]\!]$ is then defined as the set of sequential executions w which can be derived from the empty word:

$$\epsilon = w_0 \xrightarrow{R_{i_1}} w_1 \xrightarrow{R_{i_2}} w_2 \ldots \xrightarrow{R_{i_p}} w_p = w,$$

where $i_1, \ldots, i_p$ is a non-decreasing sequence of integers from $\{1 \ldots, n\}$. This means that the rules must be applied in order, and each rule can be applied 0 or several times.

Below we give inductive definitions for the atomic queue and stack data structures. Other data structures such as atomic registers and mutexes also have inductive definitions, as demonstrated in the extended version of this paper [5].

Example 5. The queue has a method *Enq* to add an element to the data structure, and a method *Deq* to remove the elements in a FIFO order. The method *DeqEmpty* can only return when the queue is empty (its parameter is not used). The only input method is *Enq*. Formally, Queue is defined by the rules R_0, R_{Enq}, R_{EnqDeq} and $R_{DeqEmpty}$.

$$\begin{aligned}
R_0 &\equiv \epsilon \in \textsf{Queue} \\
R_{Enq} &\equiv u \in \textsf{Queue} \land u \in Enq^* \Rightarrow u \cdot Enq \in \textsf{Queue} \\
R_{EnqDeq} &\equiv u \cdot v \in \textsf{Queue} \land u \in Enq^* \land v \in \{Enq, Deq\}^* \Rightarrow Enq \cdot u \cdot Deq \cdot v \in \textsf{Queue} \\
R_{DeqEmpty} &\equiv u \cdot v \in \textsf{Queue} \land \textsf{matched}(Enq, u) \Rightarrow u \cdot DeqEmpty \cdot v \in \textsf{Queue}
\end{aligned}$$

One derivation for Queue is:

$$\begin{aligned}
\epsilon \in \textsf{Queue} &\xrightarrow{R_{EnqDeq}} Enq(1) \cdot Deq(1) \in \textsf{Queue} \\
&\xrightarrow{R_{EnqDeq}} Enq(2) \cdot Enq(1) \cdot Deq(2) \cdot Deq(1) \in \textsf{Queue} \\
&\xrightarrow{R_{EnqDeq}} Enq(3) \cdot Deq(3) \cdot Enq(2) \cdot Enq(1) \cdot Deq(2) \cdot Deq(1) \in \textsf{Queue} \\
&\xrightarrow{R_{DeqEmpty}} Enq(3) \cdot Deq(3) \cdot DeqEmpty \cdot Enq(2) \cdot Enq(1) \cdot Deq(2) \cdot Deq(1) \in \textsf{Queue}
\end{aligned}$$

Similarly, Stack is composed of the rules $R_0, R_{PushPop}, R_{Push}, R_{PopEmpty}$.

$$\begin{aligned}
R_0 &\equiv \epsilon \in \textsf{Stack} \\
R_{PushPop} &\equiv u \cdot v \in \textsf{Stack} \land \textsf{matched}(Push, u) \land \textsf{matched}(Push, v) \land u, v \in \{Push, Pop\}^* \\
&\qquad \Rightarrow Push \cdot u \cdot Pop \cdot v \in \textsf{Stack} \\
R_{Push} &\equiv u \cdot v \in \textsf{Stack} \land \textsf{matched}(Push, u) \land u, v \in \{Push, Pop\}^* \Rightarrow u \cdot Push \cdot v \in \textsf{Stack} \\
R_{PopEmpty} &\equiv u \cdot v \in \textsf{Stack} \land \textsf{matched}(Push, u) \Rightarrow u \cdot PopEmpty \cdot v \in \textsf{Stack}
\end{aligned}$$

We assume that the rules defining a data structure $\mathcal{S}$ satisfy a non-ambiguity property stating that the last step in deriving a sequential execution in $[\![\mathcal{S}]\!]$ is unique and it can be effectively determined. Since we are interested in characterizing the linearizations of a history and its projections, this property is extended to permutations of projections of sequential executions which are admitted by $\mathcal{S}$. Thus, we assume that the rules defining a data structure are *non-ambiguous*, that is:

– for all $u \in [\![\mathcal{S}]\!]$, there exists a unique rule, denoted by $\texttt{last}(u)$, that can be used as the last step to derive u, i.e., for every sequence of rules $R_{i_1}, \ldots, R_{i_n}$ leading to u, $R_{i_n} = \texttt{last}(u)$. For $u \notin [\![\mathcal{S}]\!]$, $\texttt{last}(u)$ is also defined but can be arbitrary, as there is no derivation for u.

– if $\texttt{last}(u) = R_i$, then for every permutation $u' \in [\![\mathcal{S}]\!]$ of a projection of u, $\texttt{last}(u') = R_j$ with $j \leq i$. If u' is a permutation of u, then $\texttt{last}(u') = R_i$.

Given a (completed) history h, all the u such that $h \sqsubseteq u$ are permutations of one another. The last condition of non-ambiguity thus enables us to extend the function `last` to histories: $\texttt{last}(h)$ is defined as $\texttt{last}(u)$ where u is any sequential execution such that $h \sqsubseteq u$. We say that $\texttt{last}(h)$ is the rule *corresponding* to h.

Example 6. For Queue, we define `last` for a sequential execution u as follows:

– if u contains a *DeqEmpty* operation, $\texttt{last}(u) = R_{DeqEmpty}$,
– else if u contains a *Deq* operation, $\texttt{last}(u) = R_{EnqDeq}$,
– else if u contains only *Enq*'s, $\texttt{last}(u) = R_{Enq}$,
– else (if u is empty), $\texttt{last}(u) = R_0$.

Since the conditions we use to define `last` are closed under permutations, we get that for any permutation u_2 of u, $\texttt{last}(u) = \texttt{last}(u_2)$, and `last` can be extended to histories. Therefore, the rules $R_0, R_{EnqDeq}, R_{DeqEmpty}$ are non-ambiguous.

4 Reducing Linearizability to State Reachability

Our end goal for this section is to show that for any data-independent implementation $\mathcal{I}$, and any specification $\mathcal{S}$ satisfying several conditions defined in the following, there exists a computable finite-state automaton $\mathcal{A}$ (over call and return actions) such that:

$$\mathcal{I} \sqsubseteq \mathcal{S} \iff \mathcal{I} \cap \mathcal{A} = \emptyset$$

Then, given a model of $\mathcal{I}$, the linearizability of $\mathcal{I}$ is reduced to checking emptiness of the synchronized product between the model of $\mathcal{I}$ and $\mathcal{A}$. The automaton $\mathcal{A}$ represents (a subset of the) executions which are not linearizable with respect to $\mathcal{S}$.

The first step in proving our result is to show that, under some conditions, we can partition the concurrent executions which are not linearizable with respect to $\mathcal{S}$ into a finite number of classes. Intuitively, each non-linearizable execution must correspond to a violation for one of the rules in the definition of $\mathcal{S}$.

We identify a property, which we call *step-by-step linearizability*, which is sufficient to obtain this characterization. Intuitively, step-by-step linearizability enables us to build a linearization for an execution e incrementally, using linearizations of projections of e.

The second step is to show that, for each class of violations (i.e., with respect to a specific rule R_i), we can build a regular automaton $\mathcal{A}_i$ such that: a) when restricted to well-formed executions, $\mathcal{A}_i$ recognizes a subset of this class; b) each non-linearizable execution has a corresponding execution, obtained by data independence, accepted by $\mathcal{A}_i$. If such an automaton exists, we say that R_i is *co-regular* (formally defined later in this section).

We prove that, provided these two properties hold, we have the equivalence mentioned above, by defining $\mathcal{A}$ as the union of the $\mathcal{A}_i$'s built for each rule R_i.

4.1 Reduction to a Finite Number of Classes of Violations

Our goal here is to give a characterization of the sequential executions which belong to a data structure, as well as to give a characterization of the concurrent executions which are linearizable with respect to the data structure. This characterization enables us to classify the linearization violations into a finite number of classes.

Our characterization relies heavily on the fact that the data structures we consider are *closed under projection*, i.e., for all $u \in \mathcal{S}, D \subseteq \mathbb{D}$, we have $u_{|D} \in \mathcal{S}$. The reason for this is that the guards used in the inductive rules are closed under projection.

Lemma 2. *Any data structure $\mathcal{S}$ defined in our framework is closed under projection.*

A sequential execution u is said to *match* a rule R with conditions *Guard* if there exist a data value x and sequential executions $u_1, \ldots, u_k$ such that u can be written as $[\![Expr(u_1, \ldots, u_k)]\!]$, where x is the data value used for the method events, and such that $Guard(u_1, \ldots, u_k)$ holds. We call x the *witness* of the decomposition. We denote by MR the set of sequential executions which match R, and we call it the *matching set* of R.

Example 7. MR_{EnqDeq} is the set of sequential executions of the form $Enq(x) \cdot u \cdot Deq(x) \cdot v$ for some $x \in \mathbb{D}$, and with $u \in Enq^*$.

Lemma 3. *Let $\mathcal{S} = R_1, \ldots, R_n$ be a data structure and u a differentiated sequential execution. Then,*

$$u \in \mathcal{S} \iff \mathsf{proj}(u) \subseteq \bigcup_{i \in \{1,\ldots,n\}} MR_i$$

This characterization enables us to get rid of the recursion, so that we only have to check non-recursive properties. We want a similar lemma to characterize $e \sqsubseteq \mathcal{S}$ for an execution e. This is where we introduce the notion of *step-by-step linearizability*, as the lemma will hold under this condition.

Definition 7. *A data structure $\mathcal{S} = R_1, \ldots, R_n$ is said be to* step-by-step linearizable *if for any differentiated execution e, if e is linearizable w.r.t. MR_i with witness x, we have:*

$$e \setminus x \sqsubseteq [\![R_1, \ldots, R_i]\!] \implies e \sqsubseteq [\![R_1, \ldots, R_i]\!]$$

This notion applies to the usual data structures, as shown by the following lemma. The generic schema we use is the following: we let $u' \in [\![R_1, \ldots, R_i]\!]$ be a sequential execution such that $e \setminus x \sqsubseteq u'$ and build a graph G from u', whose acyclicity implies that $e \sqsubseteq [\![R_1, \ldots, R_i]\!]$. Then, we show that we can always choose u' so that G is acyclic.

Lemma 4. Queue, Stack, Register, *and* Mutex *are step-by-step linearizable.*

Intuitively, step-by-step linearizability will help us prove the right-to-left direction of Lemma 5 by allowing us to build a linearization for e incrementally, from the linearizations of projections of e.

Lemma 5. *Let $\mathcal{S}$ be a data structure with rules $R_1, \ldots, R_n$. Let e be a differentiated execution. If $\mathcal{S}$ is step-by-step linearizable, we have (for any j):*

$$e \sqsubseteq [\![R_1, \ldots, R_j]\!] \iff \mathsf{proj}(e) \sqsubseteq \bigcup_{i \leq j} MR_i$$

Thanks to Lemma 5, if we're looking for an execution e which is not linearizable w.r.t. some data-structure $\mathcal{S}$, we must prove that $\mathsf{proj}(e) \not\sqsubseteq \bigcup_i MR_i$, i.e., we must find a projection $e' \in \mathsf{proj}(e)$ which is not linearizable with respect to any MR_i ($e' \not\sqsubseteq \bigcup_i MR_i$).

This is challenging as it is difficult to check that an execution is not linearizable w.r.t. a union of sets simultaneously. Using non-ambiguity, we simplify this check by making it more modular, so that we only have to check one set MR_i at a time.

Lemma 6. *Let $\mathcal{S}$ be a data structure with rules $R_1, \ldots, R_n$. Let e be a differentiated execution. If $\mathcal{S}$ is step-by-step linearizable, we have:*

$$e \sqsubseteq \mathcal{S} \iff \forall e' \in \mathsf{proj}(e).\ e' \sqsubseteq MR \text{ where } R = \mathtt{last}(e')$$

Lemma 6 gives us the finite kind of violations that we mentioned in the beginning of the section. More precisely, if we negate both sides of the equivalence, we have: $e \not\sqsubseteq \mathcal{S} \iff \exists e' \in \mathsf{proj}(e).\ e' \not\sqsubseteq MR$. This means that whenever an execution is not linearizable w.r.t. $\mathcal{S}$, there can be only finitely reasons, namely there must exist a projection which is not linearizable w.r.t. the matching set of its corresponding rule.

4.2 Regularity of Each Class of Violations

Our goal is now to construct, for each R, an automaton $\mathcal{A}$ which recognizes (a subset of) the executions e, which have a projection e' such that $e' \not\sqsubseteq MR$. More precisely, we want the following property.

Definition 8. *A rule R is said to be* co-regular *if we can build an automaton $\mathcal{A}$ such that, for any data-independent implementation $\mathcal{I}$, we have:*

$$\mathcal{A} \cap \mathcal{I} \neq \emptyset \iff \exists e \in \mathcal{I}_{\neq}, e' \in \mathsf{proj}(e).\ \mathtt{last}(e') = R \wedge e' \not\sqsubseteq MR$$

A data structure $\mathcal{S}$ is co-regular *if all of its rules are co-regular.*

Formally, the alphabet of $\mathcal{A}$ is $\{\mathtt{call}\ m(x) \mid m \in \mathbb{M}, x \in D\} \cup \{\mathtt{ret}\ m(x) \mid m \in \mathbb{M}, x \in D\}$ for a finite subset $D \subseteq \mathbb{D}$. The automaton doesn't read operation identifiers, thus, when taking the intersection with $\mathcal{I}$, we ignore them.

Lemma 7. Queue*,* Stack*,* Register*, and* Mutex *are co-regular.*

Proof. To illustrate this lemma, we sketch the proof for the rule $R_{DeqEmpty}$ of Queue. The complete proof of the lemma can be found in the extended version of this paper.

We prove in the extended version that a history has a projection such that $\mathtt{last}(h') = R_{DeqEmpty}$ and $h' \not\sqsubseteq MR_{DeqEmpty}$ if and only if it has a *DeqEmpty* operation which is *covered* by other operations, as depicted in Fig. 1. The automaton $\mathcal{A}_{R_{DeqEmpty}}$ in Fig. 2 recognizes such violations.

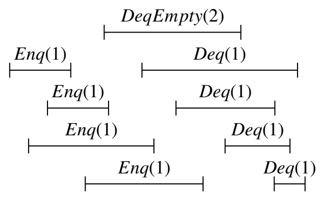

Fig. 1. A four-pair $R_{DeqEmpty}$ violation. The extended version of this paper demonstrates that this pattern with arbitrarily-many pairs is regular.

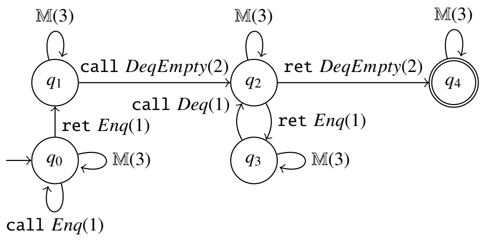

Fig. 2. An automaton recognizing $R_{DeqEmpty}$ violations, for which the queue is non-empty, with data value 1, for the span of *DeqEmpty*. We assume all `call` *Enq*(1) actions occur initially without loss of generality due to implementations' closure properties.

Let $\mathcal{I}$ be any data-independent implementation. We show that

$$\mathcal{A}_{R_{DeqEmpty}} \cap \mathcal{I} \neq \emptyset \iff \exists e \in \mathcal{I}_{\neq}, e' \in \mathsf{proj}(e).\ \mathtt{last}(e') = R_{DeqEmpty} \wedge e' \not\sqsubseteq MR_{DeqEmpty}$$

($\Rightarrow$) Let $e \in \mathcal{I}$ be an execution which is accepted by $\mathcal{A}_{R_{DeqEmpty}}$. By data independence, let $e_{\neq} \in \mathcal{I}$ and r a renaming such that $e = r(e_{\neq})$. Let $d_1, \ldots, d_m$ be the data values which are mapped to value 1 by r.

Let d be the data value which is mapped to value 2 by r. Let o the *DeqEmpty* operation with data value d. By construction of the automaton we can prove that o is covered by $d_1, \ldots, d_m$, and conclude that h has a projection such that $\mathtt{last}(h') = R_{DeqEmpty}$ and $h' \not\sqsubseteq MR_{DeqEmpty}$.

($\Leftarrow$) Let $e_{\neq} \in \mathcal{I}_{\neq}$ such that there is a projection e' such that $\mathtt{last}(e') = R_{DeqEmpty}$ and $e' \not\sqsubseteq MR_{DeqEmpty}$. Let $d_1, \ldots, d_m$ be the data values given by the $R_{DeqEmpty}$-characterization in the full version of this paper, and let d be the data value corresponding to the *DeqEmpty* operation.

Without loss of generality, we can always choose the cycle so that $Enq(d_i)$ doesn't happen before $Deq(d_{i-2})$ (if it does, drop d_{i-1}).

Let r be the renaming which maps $d_1, \ldots, d_m$ to 1, d to 2, and all other values to 3. Let $e = r(e_{\neq})$. The execution e can be recognized by automaton $\mathcal{A}_{R_{DeqEmpty}}$, and belongs to $\mathcal{I}$ by data independence.

When we have a data structure which is both step-by-step linearizable and co-regular, we can make a linear time reduction from the verification of linearizability with respect to $\mathcal{S}$ to a reachability problem, as illustrated in Theorem 1.

Theorem 1. *Let $\mathcal{S}$ be a step-by-step linearizable and co-regular data structure and let $\mathcal{I}$ be a data-independent implementation. There exists a regular automaton $\mathcal{A}$ such that:*

$$\mathcal{I} \sqsubseteq \mathcal{S} \iff \mathcal{I} \cap \mathcal{A} = \emptyset$$

5 Decidability and Complexity of Linearizability

Theorem 1 implies that the linearizability problem with respect to any step-by-step linearizable and co-regular specification is decidable for any data-independent implementation for which checking the emptiness of the intersection with finite-state automata is decidable. Here, we give a class C of data-independent implementations for which the latter problem, and thus linearizability, is decidable.

Each method of an implementation in C manipulates a finite number of local variables which store Boolean values, or data values from $\mathbb{D}$. Methods communicate through a finite number of shared variables that also store Boolean values, or data values from $\mathbb{D}$. Data values may be assigned, but never used in program predicates (e.g., in the conditions of `if` and `while` statements) so as to ensure data independence. This class captures typical implementations, or finite-state abstractions thereof, e.g., obtained via predicate abstraction.

Let $\mathcal{I}$ be an implementation from class C. The automata $\mathcal{A}$ constructed in the proof of Lemma 7 use only data values 1, 2, and 3. Checking emptiness of $\mathcal{I} \cap \mathcal{A}$ is thus equivalent to checking emptiness of $\mathcal{I}_3 \cap \mathcal{A}$ with the three-valued implementation $\mathcal{I}_3 = \{e \in \mathcal{I} \mid e = e_{|\{1,2,3\}}\}$. The set $\mathcal{I}_3$ can be represented by a Petri net since bounding data values allows us to represent each thread with a finite-state machine. Intuitively, each token in the Petri net represents another thread. The number of threads can be unbounded since the number of tokens can. Places count the number of threads in each control location, which includes a local-variable valuation. Each shared variable also has one place per value to store its current valuation.

Emptiness of the intersection with regular automata reduces to the EXPSPACE-complete coverability problem for Petri nets. Limiting verification to a bounded number of threads lowers the complexity of coverability to PSPACE [7]. The hardness part of Theorem 2 comes from the hardness of state reachability in finite-state concurrent programs.

Theorem 2. *Verifying linearizability of an implementation in C with respect to a step-by-step linearizable and co-regular specification is PSPACE-complete for a fixed number of threads, and EXPSPACE-complete otherwise.*

6 Related Work

Several works investigate the theoretical limits of linearizability verification. Verifying a single execution against an arbitrary ADT specification is NP-complete [9]. Verifying all executions of a finite-state implementation against an arbitrary ADT specification (given as a regular language) is EXPSPACE-complete when program threads are bounded [2, 10], and undecidable otherwise [3].

Existing automated methods for proving linearizability of an atomic object implementation are also based on reductions to safety verification [1, 11, 13]. Vafeiadis [13] considers implementations where operation's *linearization points* are fixed to particular source-code locations. Essentially, this approach instruments the implementation with ghost variables simulating the ADT specification at linearization points. This approach is incomplete since not all implementations have fixed linearization points. Aspect-oriented proofs [11] reduce linearizability to the verification of four simpler safety properties. However, this approach has only been applied to queues, and has not produced a fully automated and complete proof technique. Dodds et al. [6] prove linearizability of stack implementations with an automated proof assistant. Their approach does not lead to full automation however, e.g., by reduction to safety verification.

7 Conclusion

We have demonstrated a linear-time reduction from linearizability for fixed ADT specifications to control-state reachability, and the application of this reduction to atomic queues, stacks, registers, and mutexes. Besides yielding novel decidability results, our reduction enables the use of existing safety-verification tools for linearizability. While this work only applies the reduction to these four objects, our methodology also applies to other typical atomic objects including semaphores and sets. Although this methodology currently does not capture priority queues, which are not data independent, we believe our approach can be extended to include them. We leave this for future work.

References

[1] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS '13*. Springer, 2013.

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

[3] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Verifying concurrent programs against sequential specifications. In *ESOP '13*. Springer, 2013.

[4] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Tractable refinement checking for concurrent objects. In *POPL '15*. ACM, 2015.

[5] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. On reducing linearizability to state reachability. *CoRR*, abs/1502.06882, 2015. URL `http://arxiv.org/abs/1502.06882`.

[6] M. Dodds, A. Haas, and C. M. Kirsch. A scalable, correct time-stamped stack. In *POPL '15*. ACM, 2015.

[7] J. Esparza. Decidability and complexity of petri net problems — an introduction. In *Lectures on Petri Nets I: Basic Models*. Springer Berlin Heidelberg, 1998.

[8] I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52), 2010.

[9] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM J. Comput.*, 26(4), 1997.

[10] J. Hamza. On the complexity of linearizability. *CoRR*, abs/1410.5000, 2014. URL `http://arxiv.org/abs/1410.5000`.

[11] T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR '13*. Springer, 2013.

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

[13] V. Vafeiadis. Automatically proving linearizability. In *CAV '10*. Springer, 2010.