On Reducing Linearizability to State Reachability[1]

Ahmed Bouajjani[a], Michael Emmi[b], Constantin Enea[a], Jad Hamza[a]

[a]*LIAFA, Université Paris Diderot*
[b]*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 [11], being an equivalent property [7, 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. An ADT is an object whose logical behavior is defined by a set of operations.

Verifying automatically 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

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

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 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 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 7.

2. Preliminaries

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.[2] We identify a subset $\mathbb{M}_i \subseteq \mathbb{M}$ of *input* methods in order to differentiate methods taking an argument (e.g., the *Enq* method which inserts the argument value into a queue) from the other methods (e.g., the *Deq* method which doesn't take an argument, and returns the first element of a queue).

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.*

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 $\texttt{call}_{\texttt{o}}\ m(x)$. Similarly, a *return action* is denoted $\texttt{ret}_{\texttt{o}}\ m(x)$. 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 1. $\texttt{call}_{\texttt{o}_1}\ Enq(7) \cdot \texttt{call}_{\texttt{o}_2}\ Enq(4) \cdot \texttt{ret}_{\texttt{o}_1}\ Enq(7) \cdot \texttt{ret}_{\texttt{o}_2}\ Enq(4)$ *is an execution, while* $\texttt{call}_{\texttt{o}_1}\ Enq(7) \cdot \texttt{call}_{\texttt{o}_2}\ Enq(4) \cdot \texttt{ret}_{\texttt{o}_1}\ Enq(7) \cdot \texttt{ret}_{\texttt{o}_1}\ Enq(4)$ *and* $\texttt{call}_{\texttt{o}_1}\ Enq(7) \cdot \texttt{ret}_{\texttt{o}_1}\ Enq(7) \cdot \texttt{ret}_{\texttt{o}_2}\ Enq(4)$ *are not.*

[2]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.

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 $e \cdot c \cdot a \cdot e' \in \mathcal{I}$.

- Returns can be 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 [10]: 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 simplify 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 $\texttt{call}_o\ m(x)$ is labeled by (m, x).

Example 2. *The history of the execution*

$$\texttt{call}_{o_1}\ \mathit{Enq}(7) \cdot \texttt{call}_{o_2}\ \mathit{Enq}(4) \cdot \texttt{ret}_{o_1}\ \mathit{Enq}(7) \cdot \texttt{ret}_{o_2}\ \mathit{Enq}(4)$$

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 $f : O \to \{1, \ldots, n\}$ 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 S of sequential executions, denoted $h \sqsubseteq S$, if there exists $u \in S$ such that $h \sqsubseteq u$.*

A set of histories H is *linearizable* with respect to S, denoted $H \sqsubseteq S$ if $h \sqsubseteq S$ for all $h \in H$. We extend these definitions to executions according to their histories. In that context, the set S is called a *specification*.

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

Example 3. $\texttt{call}_{\texttt{o}_1}\ \mathit{Enq}(7) \cdot \texttt{call}_{\texttt{o}_2}\ \mathit{Enq}(7) \cdot \texttt{ret}_{\texttt{o}_1}\ \mathit{Enq}(7) \cdot \texttt{ret}_{\texttt{o}_2}\ \mathit{Enq}(7)$ *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) S is* data independent *if:*

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

When checking that a data-independent implementation $\mathcal{I}$ is linearizable with respect to a data-independent specification 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 S, if and only if $\mathcal{I}_{\neq}$ is linearizable with respect to $S_{\neq}$.*

Proof. ($\Rightarrow$) Let e be a (differentiated) execution in $\mathcal{I}_{\neq}$. By assumption, it is linearizable with respect to a sequential execution u in S, and the bijection between the operations of e and the method events of u, ensures that u is differentiated and belongs to $S_{\neq}$.

($\Leftarrow$) Let e be an execution in $\mathcal{I}$. By data independence of $\mathcal{I}$, we know there exists $e_{\neq} \in \mathcal{I}_{\neq}$ and a renaming r such that $r(e_{\neq}) = e$. By assumption, $e_{\neq}$ is linearizable with respect to a sequential execution $u_{\neq} \in S_{\neq}$. We define $u = r(u_{\neq})$, and know by data independence of S that $u \in S$. Moreover, we can use the same bijection used for $e_{\neq} \sqsubseteq u_{\neq}$ to prove that $e \sqsubseteq u$. □

3. Inductively-Defined Data Structures

A *data-structure* S is given as an ordered sequence of rules $R_1, \dots, R_n$, each of the form: $u_1 \cdot u_2 \cdots u_k \in S \wedge Guard(u_1, \dots, u_k) \Rightarrow Expr(u_1, \dots, u_k) \in S$, where the variables u_i are interpreted over sequential executions, and

- $Guard(u_1, \dots, u_k)$ is a conjunction of conditions on $u_1, \dots, u_k$ with atoms
  - $u_i \in \mathbb{M}^*$ $(M \subseteq \mathbb{M})$
  - $\mathsf{matched}(m, u_i)$
- $Expr(u_1, \dots, u_k)$ is an *expression* $e = a_1 \cdot a_2 \cdots a_l$ where
  - $u_1, \dots, u_k$ 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 $\epsilon \in S$.

A condition $u_i \in \mathbb{M}^*$ $(M \subseteq \mathbb{M})$ is satisfied when the methods used in u_i are all in $\mathbb{M}$. 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 $u = u_1 \cdot \ldots \cdot u_k$ and an expression $e = Expr(u_1, \dots, u_k)$, 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 $R \equiv u_1 \cdot u_2 \cdots u_k \in S \wedge Guard(u_1, \dots, u_k) \Rightarrow Expr(u_1, \dots, u_k) \in S$ is applied to a sequential execution w to obtain a new sequential execution w' from the set:

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

We denote this $w \xrightarrow{R} w'$. The set of sequential executions $[\![S]\!] = [\![R_1, \dots, R_n]\!]$ 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.

When clear from context, we abuse notation and denote the set of sequential executions $[\![S]\!]$ by S. Below we give inductive definitions for the atomic queue, stack, register, and mutex data-structures.

Example 4. *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* $R_{DeqEmpty}$.

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

One derivation for Queue *is:*

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

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

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

The register has a method Write used to write a data value, and a method Read which returns the last written value. The only input method is Write. Its rules are R_0 *and* R_{WR}:

$$\begin{aligned}
R_0 &\equiv \epsilon \in \mathsf{Register}\\
R_{WR} &\equiv u \in \mathsf{Register} \Rightarrow Write \cdot Read^* \cdot u \in \mathsf{Register}
\end{aligned}$$

The mutex has a method Lock, used to take ownership of the Mutex, *and a method Unlock, to release it. The only input method is Lock. It is composed of the rules* R_0, R_{Lock} *and* R_{LU}:

$$\begin{aligned}
R_0 &\equiv \epsilon \in \mathsf{Mutex}\\
R_{Lock} &\equiv Lock \in \mathsf{Mutex}\\
R_{LU} &\equiv u \in \mathsf{Mutex} \Rightarrow Lock \cdot Unlock \cdot u \in \mathsf{Mutex}
\end{aligned}$$

In practice, Lock and Unlock methods do not have a parameter. Here, the parameter represents a ghost variable *which helps us relate Unlock to their* *corresponding Lock. Any implementation will be data independent with respect to these ghost variables.*

We assume that the rules defining a data-structure S satisfy a non-ambiguity property stating that the last step in deriving a sequential execution in $\llbracket S \rrbracket$ 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 S.

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 5. *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 assume that the rules defining a data-structure are *well-formed*, that is:

- for all $u \in \llbracket S \rrbracket$, 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 $u \notin \llbracket S \rrbracket$, $\texttt{last}(u)$ is also defined but can be arbitrary, as there is no derivation for u.
- if $\texttt{last}(u) = R_i$, then for every permutation $u' \in \llbracket S \rrbracket$ 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 well-formed.*

Definition of `last` *for a sequential execution* $u \in$ Stack*:*

- *if u contains a PopEmpty operation,* $\texttt{last}(u) = R_{PopEmpty}$,

- *else if u contains an unmatched Push operation,* $\mathtt{last}(u) = R_{Push}$*,*
- *else if u contains a Pop operation,* $\mathtt{last}(u) = R_{PushPop}$*,*
- *else (if u is empty),* $\mathtt{last}(u) = R_0$*.*

Definition of `last` *for a sequential execution* $u \in \mathsf{Register}$*:*

- *if u is not empty,* $\mathtt{last}(u) = R_{WR}$*,*
- *else,* $\mathtt{last}(u) = R_0$*.*

Definition of `last` *for a sequential execution* $u \in \mathsf{Mutex}$*:*

- *if u contains an Unlock operation,* $\mathtt{last}(u) = R_{LU}$*,*
- *else if u is not empty,* $\mathtt{last}(u) = R_{Lock}$*,*
- *else,* $\mathtt{last}(u) = R_0$*.*

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 S satisfying several conditions defined in the following, there exists a computable finite-state automaton $\mathcal{A}$ (over call and return actions) such that:

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

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 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 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 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 finite 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 $u \in S, D \subseteq \mathbb{D}$, we have $u_{|D} \in S$. The reason for this is that the guards used in the inductive rules are closed under projection.

Lemma 2. *Any data-structure S defined in our framework is closed under projection.*

Proof. Let $u \in S$ and let $D \subseteq \mathbb{D}$. Since $u \in S$, there is a sequence of applications of rules starting from the empty word ϵ which can derive u. We remove from this derivation all the rules corresponding to a data value $x \notin D$, and we project all the sequential executions appearing in the derivation on the set D. Since the predicates which appear in the conditions are all closed under projection, the derivation remains valid, and proves that $u_{|D} \in S$. □

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

Example 7. $\mathsf{MS}(R_{EnqDeq})$ *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* $u \in Enq^*$*.*

When an execution e is linearizable with respect to $\mathsf{MS}(R)$, i.e. to a sequence u which matches R with some witness x, we say that e is linearizable with respect to $\mathsf{MS}(R)$ with *witness* x.

Lemma 3. *Let $S = R_1, \dots, R_n$ be a data-structure and u be a differentiated sequential execution. Then,*

$$u \in S \iff \mathsf{proj}(u) \subseteq \bigcup_{i \in \{1,\dots,n\}} \mathsf{MS}(R_i)$$

Proof. ($\Rightarrow$) Using Lemma 2, we know that S is closed under projection. Thus, any projection of a sequential execution u of S is itself in S and has to match one of the rules $R_1, \dots, R_n$.

($\Leftarrow$) By induction on the size of u. We know $u \in \mathsf{proj}(u)$, so it can be decomposed to satisfy the conditions *Guard* of some rule R of S. The recursive condition is then verified by induction. □

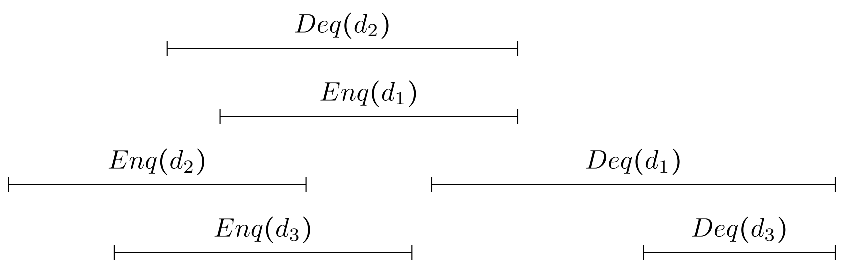


Figure 1: An execution linearizable with respect to $\mathsf{MS}(R_{EnqDeq})$ with witness d_1, and such that $e \setminus d_1$ is linearizable with respect to $[\![R_0, R_{Enq}, R_{EnqDeq}]\!]$. Step-by-step linearizability ensures that e is itself linearizable with respect to $[\![R_0, R_{Enq}, R_{EnqDeq}]\!]$. Time goes from left to right.

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 $e \sqsubseteq S$ 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 $S = R_1, \dots, R_n$ is said to be* step-by-step linearizable *if for any differentiated execution e, any $i \in \{1, \dots, n\}$ and $x \in \mathbb{D}$, if e is linearizable with respect to $\mathsf{MS}(R_i)$ with witness x, we have:*

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

Example 8. *The execution e of Fig 1 is linearizable with respect to to $\mathsf{MS}(R_{EnqDeq})$ with witness d_1. This means that $Enq(d_1)$ is minimal amongst all operations and that $Deq(d_1)$ is minimal amongst all dequeue operations. The rest of the execution $e \setminus d_1$ is linearizable with respect to $[\![R_0, R_{Enq}, R_{EnqDeq}]\!]$ (into $Enq(d_2) \cdot Enq(d_3) \cdot Deq(d_2) \cdot Deq(d_3)$). The notion of step-by-step linearizability ensures that e is also linearizable with respect to $[\![R_0, R_{Enq}, R_{EnqDeq}]\!]$ (into $Enq(d_1) \cdot Enq(d_2) \cdot Enq(d_3) \cdot Deq(d_1) \cdot Deq(d_2) \cdot Deq(d_3)$).*

This notion applies to the usual data-structures, as we will prove in Section 5. Intuitively, step-by-step linearizability will help us prove the right-to-left direction of Lemma 4 by allowing us to build a linearization for e incrementally, from the linearizations of projections of e. Lemma 4 is a key step in our reasoning, and can be seen as a generalization of Lemma 3 from sequential to concurrent executions.

Lemma 4. *Let S be a data-structure with rules $R_1, \dots, R_n$. Let e be a differentiated execution. If S is step-by-step linearizable, we have (for any j):*

$$e \sqsubseteq [\![R_1, \dots, R_j]\!] \iff \mathsf{proj}(e) \sqsubseteq \bigcup_{i \le j} \mathsf{MS}(R_i)$$

Proof. ($\Rightarrow$) We know there exists $u \in [\![R_1, \dots, R_j]\!]$ such that $e \sqsubseteq u$. Each projection e' of e can be linearized with respect to some projection u' of u, which belongs to $\bigcup_{i \le j} \mathsf{MS}(R_i)$ according to Lemma 3.

($\Leftarrow$) By induction on the size of e. We know $e \in \mathsf{proj}(e)$ so it can be linearized with respect to a sequential execution u matching some rule R_k ($k \leq j$) with some witness x. Let $e' = e \setminus x$.

Since S is well-formed, we know that no projection of e can be linearized to a matching set $\mathsf{MS}(R_i)$ with $i > k$, and in particular no projection of e'. Thus, we deduce that $\mathsf{proj}(e') \sqsubseteq \bigcup_{i \leq k} \mathsf{MS}(R_i)$, and conclude by induction that $e' \sqsubseteq [\![R_1, \dots, R_k]\!]$.

We finally use the fact that S is step-by-step linearizable to deduce that $e \sqsubseteq [\![R_1, \dots, R_k]\!]$ and $e \sqsubseteq [\![R_1, \dots, R_j]\!]$ because $k \leq j$. □

Using Lemma 4, if we're looking for an execution e which is not linearizable with respect to some step-by-step linearizable data-structure S, we must prove that $\mathsf{proj}(e) \not\sqsubseteq \bigcup_i \mathsf{MS}(R_i)$, i.e. we must find a projection $e' \in \mathsf{proj}(e)$ which is not linearizable with respect to any $\mathsf{MS}(R_i)$ ($e' \not\sqsubseteq \bigcup_i \mathsf{MS}(R_i)$).

This is challenging as it is difficult to check that an execution is not linearizable with respect to 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 $\mathsf{MS}(R_i)$ at a time.

Lemma 5. *Let S be a data-structure with rules $R_1, \dots, R_n$. Let e be a differentiated execution. If S is step-by-step linearizable, we have:*

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

Proof. ($\Rightarrow$) Let $e' \in \mathsf{proj}(e)$. By Lemma 4, we know that e' is linearizable with respect to $\mathsf{MS}(R_i)$ for some i. Since S is well-formed, $\mathtt{last}(e')$ is the only rule such that $e' \sqsubseteq \mathsf{MS}(R)$ can hold, which ends this part of the proof.

($\Leftarrow$) Particular case of Lemma 4. □

Lemma 5 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: $e \not\sqsubseteq S \iff \exists e' \in \mathsf{proj}(e).\ e' \not\sqsubseteq \mathsf{MS}(R)$ (where $R = \mathtt{last}(e')$). This means that whenever an execution is not linearizable with respect to S, there can be only finitely reasons, namely there must exist a projection which is not linearizable with respect to the matching set of its corresponding rule.

4.2. Regularity of Each Class of Violations

Our goal is now to construct, for each R, a (non-deterministic) finite-state automaton $\mathcal{A}$ which recognizes (a subset of) the executions e, which have a projection e' such that $e' \not\sqsubseteq \mathsf{MS}(R)$. 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:*

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

A data-structure S is co-regular *if all of its rules are co-regular.*

Formally, the alphabet of $\mathcal{A}$ is the set $\mathsf{Ev}_S(D)$, defined as:

$$\{\texttt{call}\ m(x) \mid m \in \mathbb{M}, x \in D\} \cup \{\texttt{ret}\ m(x) \mid m \in \mathbb{M}, x \in D\}$$

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 and read events $\texttt{call}_o\ m(x)$ and $\texttt{ret}_o\ m(x)$ as $\texttt{call}\ m(x)$ and $\texttt{ret}\ m(x)$.)

As we will show in Section 6, all the data-structures we defined are co-regular. 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 S to a reachability problem, as illustrated in Theorem 1.

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

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

Proof. Let $\mathcal{A}_1, \dots, \mathcal{A}_n$ be the finite automata used to show that $R_1, \dots, R_n$ are co-regular, and let $\mathcal{A}$ be the (non-deterministic) union of the $\mathcal{A}_i$'s.

($\Rightarrow$) Assume there exists an execution $e \in \mathcal{I} \cap \mathcal{A}$. From the definition of "co-regular", there exists $e'' \in \mathcal{I}_{\neq}$ and $e' \in \mathsf{proj}(e'')$ such that $e' \notin \mathsf{MS}(R_i)$, where R_i is the rule corresponding to e'. By Lemma 5, e'' is not linearizable with respect to S.

($\Leftarrow$) Assume there exists an execution $e \in \mathcal{I}$ which is not linearizable with respect to S. By Lemma 1, we can assume that e is differentiated. By Lemma 5, it has a projection $e' \in \mathsf{proj}(e)$ such that $e' \notin \mathsf{MS}(R_i)$, where R_i is the rule corresponding to e'. By definition of co-regularity, this means that $\mathcal{I} \cap \mathcal{A}_i \neq \varnothing$, and that $\mathcal{I} \cap \mathcal{A} \neq \varnothing$. □

5. Step-by-step Linearizability

The goal of this section is to prove that all data-structures considered so far are step-by-step linearizable. More specifically, we want to prove, given a data-structure S with rules $R_1, \dots, R_n$, that for any differentiated history h, if h is linearizable with respect to $\mathsf{MS}(R_i)$ with witness x, we have:

$$h \setminus x \sqsubseteq [\![R_1, \dots, R_i]\!] \implies h \sqsubseteq [\![R_1, \dots, R_i]\!].$$

The proofs follow a generic schema which consists in the following: we let $u' \in [\![R_1, \dots, R_i]\!]$ be a sequential execution such that $h \setminus x \sqsubseteq u'$ and build a graph G from u', whose acyclicity implies that $h \sqsubseteq [\![R_1, \dots, R_i]\!]$. Then we show that we can always choose u' so that this G is acyclic.

Lemma 6. Queue*,* Stack*,* $\mathsf{Register}$*, and* Mutex *are step-by-step linearizable.*

Proof. For better readability we make a sublemma per data-structure. We begin by proving that Queue is step-by-step linearizable. Concerning rule R_{EnqDeq}, our goal is to prove that, if a history h has an $Enq(x)$ which is minimal (among all operations), and a corresponding $Deq(x)$ which is minimal among all Deq operations such that $h \setminus x$ is linearizable, then h is linearizable as well.

Similarly, with rule $R_{DeqEmpty}$, we will prove that if a history h is linearizable with respect to the matching set of $R_{DeqEmpty}$, i.e. it can be linearized to $u \cdot DeqEmpty \cdot v$ – with $\mathsf{matched}(Enq, u)$ – and h without $DeqEmpty$ is linearizable with respect to Queue, then h itself is linearizable with respect to Queue.

Lemma 7. Queue *is step-by-step linearizable.*

Proof. Let h be a differentiated history, and u a sequential execution such that $h \sqsubseteq u$. We abuse notation and mix labels with operations themselves, as operation labels of the form $Enq(x)$ or $Deq(x)$ are unique in a differentiated history. For instance, we will reference an (the) operation labeled by $Enq(d)$ as $Enq(d)$. We have three cases to consider:

1) u matches R_{Enq} with witness x: let $h' = h \setminus x$ and assume $h' \sqsubseteq [\![R_0, R_{Enq}]\!]$. Since u matches R_{Enq}, we know h only contain Enq operations. The set $[\![R_0, R_{Enq}]\!]$ is composed of the sequential executions formed by repeating the Enq method events, which means that $h \sqsubseteq [\![R_0, R_{Enq}]\!]$.

2) u matches R_{EnqDeq} with witness x: let $h' = h \setminus x$ and assume $h' \sqsubseteq [\![R_0, R_{Enq}, R_{EnqDeq}]\!]$. Let $u' \in [\![R_0, R_{Enq}, R_{EnqDeq}]\!]$ such that $h' \sqsubseteq u'$. We define a graph G whose nodes are the operations of h and there is an edge from operation o_1 to o_2 if either:

1. o_1 happens before o_2 in h, or
2. the method event corresponding to o_1 in u' is before the one corresponding to o_2, or
3. $o_1 = Enq(x)$ and o_2 is any other operation, or
4. $o_1 = Deq(x)$ and o_2 is any other Deq operation.

If G is acyclic, any total order compatible with G forms a sequence u_2 such that $h \sqsubseteq u_2$ and such that u_2 can be built from u' by adding $Enq(x)$ at the beginning and $Deq(x)$ before all Deq method events. Thus, $u_2 \in [\![R_0, R_{Enq}, R_{EnqDeq}]\!]$ and $h \sqsubseteq [\![R_0, R_{Enq}, R_{EnqDeq}]\!]$.

Assume that G has a cycle, and consider a cycle C of minimal size. We show that there is only one kind of cycle possible, and that this cycle can be avoided by choosing u' appropriately. Such a cycle can only contain one happens-before edge (edges of type 1), because if there were two, we could apply the interval order property to reduce the cycle. Similarly, since the order imposed by u' is a total order, it also satisfies the interval order property, meaning that C can only contain one edge of type 2.

Moreover, C can also contain only one edge of type 3, otherwise it would have to go through $Enq(x)$ more than once. Similarly, it can contain only one edge of type 4. It cannot contain a type 3 edge $Enq(x) \to o_1$ at the same time as a type 4 edge $Deq(x) \to o_2$, because we could shortcut the cycle by a type 3 edge $Enq(x) \to o_2$.

Finally, it cannot be a cycle of size 2. For instance, a type 2 edge cannot form a cycle with a type 1 edge because $h' \sqsubseteq u'$. The only form of cycles left are the two cycles of size 3 where:

- $Enq(x)$ is before o_1 (type 3), o_1 is before o_2 in u' (type 2), and o_2 happens before $Enq(x)$ (type 1): this is not possible, because h is linearizable with respect to u which matches R_{EnqDeq} with x as a witness. This means that u starts with the method event $Enq(x)$, and that no operation can happen-before $Enq(x)$ in h.

- $Deq(x)$ is before o_1 (type 4), o_1 is before o_2 in u' (type 2), and o_2 happens before $Deq(x)$ (type 1): by definition, we know that o_1 is a Deq operation; moreover, since h is linearizable with respect to u which matches R_{EnqDeq} with x as a witness, no Deq operation can happen-before $Deq(x)$ in h, and o_2 is an Enq operation. Let $d_1, d_2 \in \mathbb{D}$ such that $Deq(d_1) = o_1$ and $Enq(d_2) = o_2$.

  Since o_1 is before o_2 in u', we know that d_1 and d_2 must be different. Moreover, there is no happens-before edge from o_1 to o_2, or otherwise, by transitivity of the happens-before relation, we'd have a cycle of size 2 between o_1 and $Deq(x)$.

  Assume without loss of generality that o_1 is the rightmost Deq method event which is before o_2 in u', and let $o_2^1, \dots, o_2^s$ be the Enq method events between o_1 and o_2. There is no happens-before edge $o_1 < o_2^i$, because by applying the interval order property with the other happens-before edge $o_2 < Deq(x)$, we'd either have $o_1 < Deq(x)$ (forming a cycle of size 2) or $o_2 < o_2^i$ (not possible because $h' \sqsubseteq u'$ and o_2^i is before o_2 in u').

  Let u'_2 be the sequence u' where $Deq(d_1)$ has been moved after o_2. Since we know there is no happens-before edge from $Deq(d_1)$ to o_2^i or to o_2, we can deduce that: $h' \sqsubseteq u'_2$. Moreover, if we consider the sequence of deductions which proves that $u' \in [\![R_0, R_{Enq}, R_{EnqDeq}]\!]$, we can alter it when we insert the pair $Enq(d_1)$ and $o_1 = Deq(d_1)$ by inserting o_1 after the o_2^i's and after o_2, instead of before (the conditions of the rule R_{EnqDeq} allow it).

Cycles of size 3 of the form (type 3 or 4, type 1, type 2) are not possible as the operations $Enq(x)$ and $Deq(x)$ do not belong to u'. This concludes case 2), as we're able to choose u' so that G is acyclic, and prove that $h \sqsubseteq [\![R_0, R_{Enq}, R_{EnqDeq}]\!]$. After transforming u' to u'_2, if there are any cycles left, we can reapply the procedure. This procedure can only be applied a finite number of times, as at each step, the number of pairs (o_1, o_2) where o_1 is a dequeue operation and o_2 is an enqueue operation is getting smaller. As we move $Deq(d_1)$ to the right of (at least one) enqueue operations, this number for u'_2 is strictly smaller than for u'.

3) u matches $R_{DeqEmpty}$ with witness x: let o be the *DeqEmpty* operation corresponding to the witness. Let $h' = h \setminus x$ and assume $h' \sqsubseteq \mathsf{Queue}$. Without loss of generality, we assume that u does not contain a $Deq(d)$ operation before its corresponding $Enq(d)$ operation, for some $d \in \mathbb{D}$. The reason is that, when $Deq(d)$ is before $Enq(d)$ in u, we can move $Deq(d)$ to the right, and $Enq(d)$ to the left, until $Enq(d)$ becomes to the left of $Deq(d)$ in the sequence u, and still have $h \sqsubseteq u$. Indeed, it is not possible that there exist (possibly equal) operations o_1, o_2 such that $Deq(d)$ happens before o_1 (in h), o_2 happens before $Enq(d)$, and $Deq(d)$, o_1, o_2, $Enq(d)$ appear in that order in u. (Operation o_1 prevents $Deq(d)$ from being moved to the right, while o_2 prevents $Enq(d)$ from being moved to the left.) The happens-before relation (of h) being an interval order, that would imply either $Deq(d)$ happens before $Enq(d)$ contradicting $h' \sqsubseteq u'$, or o_2 happens before o_1 contradicting $h \sqsubseteq u$.

Let L be the set of operations which are before o in u, and R the ones which are after. Let D_L be the data values appearing in L and D_R be the data values appearing in R. Since u matches $R_{DeqEmpty}$, we know that L contains no unmatched Enq operations.

Let $u' \in \mathsf{Queue}$ such that $h' \sqsubseteq u'$. Let $u'_L = u'_{|D_L}$ and $u'_R = u'_{|D_R}$. Since Queue is closed under projection, $u'_L, u'_R \in \mathsf{Queue}$. Let $u_2 = u'_L \cdot o \cdot u'_R$. We can show that $u_2 \in \mathsf{Queue}$ by using the derivations of u'_L and u'_R. Intuitively, this is because Queue is closed under concatenation when the left-hand sequential execution has no unmatched Enq method event, like u'_L.

Moreover, we have $h \sqsubseteq u_2$, as shown in the following. We define a graph G whose nodes are the operations of h and there is an edge from operation o_1 to o_2 if either:

1. o_1 happens before o_2 in h, or
2. the method event corresponding to o_1 in u_2 is before the one corresponding to o_2.

Assume there is a cycle in G, meaning there exists o_1, o_2 such that o_1 happens before o_2 in h, but the corresponding method events are in the opposite order in u_2.

- If $o_1, o_2 \in L$, or $o_1, o_2 \in R$, this contradicts $h' \sqsubseteq u'$.
- If $o_1 \in R$ and $o_2 \in L$, this contradicts $h \sqsubseteq u$.
- If $o_1 \in R$ and $o_2 = o$, or if $o_1 = o$ and $o_2 \in L$, this contradicts $h \sqsubseteq u$.

This shows that $h \sqsubseteq u_2$ as it proves that the happens-before relation of h is a subset of the relation induced by the sequence u_2. Thus, we have $h \sqsubseteq \mathsf{Queue}$ and concludes the proof that the Queue is step-by-step linearizable. $\square$

Proving that Stack is step-by-step linearizable can be done like for the rule $R_{DeqEmpty}$ of Queue. The idea is again to combine two linearizations of subhistories into a linearization for the full history h.

Lemma 8. Stack *is step-by-step linearizable.*

Proof. Let h be a differentiated history, and u a sequential execution such that $h \sqsubseteq u$. We have three cases to consider:

1) (very similar to case 3 of the Queue) u matches $R_{PushPop}$ with witness x: let a and b be respectively the Push and Pop operations corresponding to the witness. Let $h' = h \setminus x$ and assume $h' \sqsubseteq [\![R_0,R_{PushPop}]\!]$. Let L be the set of operations which are before b in u, and R the ones which are after. Let D_L be the data values appearing in L and D_R be the data values appearing in R. Since u matches $R_{PushPop}$, we know that L contains no unmatched Push operations.

Let $u' \in [\![R_0,R_{PushPop}]\!]$ such that $h' \sqsubseteq u'$. Let $u'_L = u'_{|D_L}$ and $u'_R = u'_{|D_R}$. Since $[\![R_0,R_{PushPop}]\!]$ is closed under projection, $u'_L, u'_R \in [\![R_0,R_{PushPop}]\!]$. Let $u_2 = a \cdot u'_L \cdot b \cdot u'_R$. We can show that $u_2 \in [\![R_0,R_{PushPop}]\!]$ by using the derivations of u'_L and u'_R.

Moreover, we have $h \sqsubseteq u_2$, because if the total order of u_2 didn't respect the happens-before relation of u_2, it could only be because of four reasons, all leading to a contradiction:

- the violation is between two L operations or two R operations, contradicting $h' \sqsubseteq u'$
- the violation is between a L and an R operation, contradicting $h \sqsubseteq u$
- the violation is between b and another operation, contradicting $h \sqsubseteq u$
- the violation is between a and another operation contradicting $h \sqsubseteq u$

This shows that $h \sqsubseteq [\![R_0,R_{PushPop}]\!]$ and concludes case 1.

2) u matches R_{Push} with witness x: similar to case 1

3) u matches $R_{PopEmpty}$ with witness x: identical to case 3 of the Queue □

Proving step-by-step linearizability for Register and Mutex is similar to the R_{EnqDeq} rule of Queue.

Lemma 9. Register *is step-by-step linearizable.*

Proof. Let h be a differentiated history, and u a sequential execution such that $h \sqsubseteq u$ and such that u matches the rule R_{WR} with witness x. Let a and $b_1, \dots, b_s$ be respectively the *Write* and *Read*'s operations of h corresponding to the witness.

Let $h' = h \setminus x$ and assume $h' \sqsubseteq [\![R_0,R_{WR}]\!]$. Let $u' \in [\![R_0,R_{WR}]\!]$ such that $h' \sqsubseteq u'$. Let $u_2 = a \cdot b_1 \cdot b_2 \cdots b_s \cdot u'$. By using rule R_{WR} on u', we have $u_2 \in [\![R_0,R_{WR}]\!]$. Moreover, we prove that $h \sqsubseteq u_2$ by contradiction. Assume that the total order imposed by u_2 doesn't respect the happens-before relation of h. All three cases are not possible:

- the violation is between two u' operations, contradicting $h' \sqsubseteq u'$,
- the violation is between a and another operation, i.e. there is an operation o which happens before a in h, contradicting $h \sqsubseteq u$,

- the violation is between some b_i and a u' operation, i.e. there is an operation o which happens before b_i in h, contradicting $h \sqsubseteq u$.

Thus, we have $h \sqsubseteq u_2$ and $h \sqsubseteq [\![R_0,R_{WR}]\!]$, which ends the proof. □

Lemma 10. Mutex *is step-by-step linearizable.*

Proof. Identical to the Register proof, except there is only one Unlock operation (b), instead of several Read operations ($b_1,\dots,b_s$). □

□

6. Co-Regularity

Our goal in this section is to prove that each rule R we considered is co-regular, meaning, we can build an automaton $\mathcal{A}$ such that, for any data-independent implementation $\mathcal{I}$, we have:

$$\mathcal{A} \cap \mathcal{I} \neq \varnothing \iff \exists e \in \mathcal{I}_{\neq}, e' \in \mathsf{proj}(e).\ \mathtt{last}(e') = R \land e' \notin \mathsf{MS}(R).$$

We have a generic schema to build the automaton, which is first to characterize a violation by the existence of a cycle of some kind, and then build an automaton recognizing such cycles. For some of the rules, we prove that these cycles can always be bounded, thanks to a *small model property.* For the others, even though the cycles can be unbounded, we can still build an automaton.

We prove the co-regularity of R_{EnqDeq} and $R_{DeqEmpty}$ respectively in Section 6.1 and Section 6.2. The two rules require different approaches, but all other rules we consider will look like one of these two. We will then explain the similarities with the rules of Stack in Section 6.3. The co-regularity of the Register and Mutex rules are similar to the co-regularity of R_{EnqDeq}.

6.1. Co-Regularity of R_{EnqDeq}

Our approach in this section is to prove a small model property for the rule R_{EnqDeq}. More precisely, we want to prove that when a history is not linearizable with respect to the matching set of R_{EnqDeq}, then it has a *small* projection which not linearizable either. We can then build an automaton which only recognizes the small violations.

Lemma 11. *Given a history h, if $\forall d_1, d_2 \in \mathbb{D}_h$, $h_{|\{d_1,d_2\}} \sqsubseteq \mathsf{MS}(R_{EnqDeq})$, then $h \sqsubseteq \mathsf{MS}(R_{EnqDeq})$.*

Note: Claims 1, 2, 3 are part of the proof of Lemma 11.

Proof. We first identify constraints which are sufficient to prove $h \sqsubseteq \mathsf{MS}(R_{EnqDeq})$.

Claim 1. *Let h be a history and d_1 a data value of $\mathbb{D}_h$. If $Enq(d_1) \nless Deq(d_1)$, and for all operations o, we have $Enq(d_1) \nless o$, and for all Deq operations o, we have $Deq(d_1) \nless o$, then h is linearizable with respect to* $\mathsf{MS}(R_{EnqDeq})$

Proof. We define a graph G whose nodes are the elements of h, and whose edges include both the happens-before relation as well as the constraints given by the Lemma. G is acyclic by assumption and any total order compatible with G corresponds to a linearization of h which is in $\mathsf{MS}(R_{EnqDeq})$.

Notice that this doesn't necessarily mean that h is linearizable with respect to the Queue, but that it's possible to linearizable h into a sequence where $Enq(d_1)$ is at the beginning and $Deq(d_1)$ is before all Deq operations. □

Given $d_1, d_2 \in \mathbb{D}_h$, we denote by $d_1 \ \mathbf{W}_{\mathbf{h},\mathsf{MS}(\mathbf{R})} \ d_2$ the fact that $h_{|\{d_1,d_2\}}$ is linearizable with respect to $\mathsf{MS}(R)$, by using d_1 as a witness. We reduce the notation to $d_1 \ \mathbf{W} \ d_2$ when the context is not ambiguous.

First, we show that if the same data value d_1 can be used as a witness for all projections on 2 data values, then we can linearize the whole history (using this same data value as a witness).

Claim 2. *For $d_1 \in \mathbb{D}_h$, if $\forall d \neq d_1$, $d_1 \ \mathbf{W} \ d$, then $h \sqsubseteq \mathsf{MS}(R_{EnqDeq})$.*

Proof. Since $\forall d \neq d_1$, $d_1 \ \mathbf{W} \ d$, the happens-before relation of h respects the constraints given by *Lemma* 1, and we can conclude that $h \sqsubseteq \mathsf{MS}(R_{EnqDeq})$. □

Next, we show the key characterization, which enables us to reduce non-linearizability with respect to $\mathsf{MS}(R_{EnqDeq})$ to the existence of a cycle in the $\not\mathbf{W}$ relation.

Claim 3. *If $h \not\sqsubseteq \mathsf{MS}(R_{EnqDeq})$, then h has a cycle $d_1 \not\mathbf{W} \ d_2 \not\mathbf{W} \ \dots \not\mathbf{W} \ d_t \not\mathbf{W} \ d_1$*

Proof. Let $d_1 \in \mathbb{D}_h$. By Lemma 2, we know there exists $d_2 \in \mathbb{D}_h$ such that $d_1 \not\mathbf{W} \ d_2$. Likewise, we know there exists $d_3 \in \mathbb{D}_h$ such that $d_2 \not\mathbf{W} \ d_3$. We continue this construction until we form a cycle. □

We can now prove the small model property stated in Lemma 11. Assume $h \not\sqsubseteq \mathsf{MS}(R)$. By Lemma 3, it has a cycle $d_1 \not\mathbf{W} \ d_2 \not\mathbf{W} \ \dots \not\mathbf{W} \ d_t \not\mathbf{W} \ d_1$. If there exists a data value x such that $Deq(x)$ happens before $Enq(x)$, then $h_{|\{x\}} \not\sqsubseteq \mathsf{MS}(R_{EnqDeq})$, which contradicts our assumptions.

For each i, there are two possible reasons for which $d_i \not\mathbf{W} \ d_{(i \ mod \ t)+1}$. The first one is that $Enq(d_i)$ is not minimal in the projection on $\{d_i, d_{(i \ mod \ t)+1}\}$ (reason (a)). The second one is that $Deq(d_i)$ is not minimal with respect to the Deq operations (reason (b)).

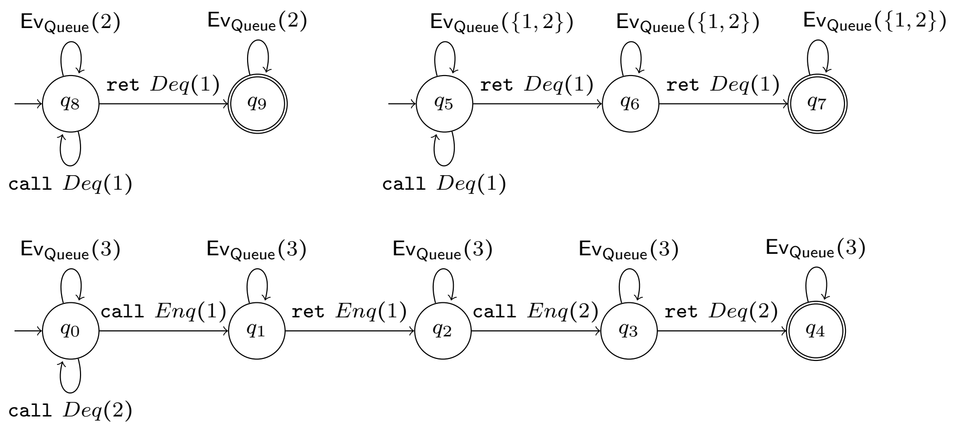
We label each edge of our cycle by either (a) or (b), depending on which one is true (if both are true, pick arbitrarily). Then, using the interval order property, we have that, if $d_i \not\mathbf{W} \ d_{(i \ mod \ t)+1}$ for reason (a), and $d_j \not\mathbf{W} \ d_{(j \ mod \ t)+1}$ for reason (a) as well, then either $d_i \not\mathbf{W} \ d_{(j \ mod \ t)+1}$, or $d_j \not\mathbf{W} \ d_{(i \ mod \ t)+1}$ (for reason (a)). This enables us to reduce the cycle and leave only one edge for reason (a).

The same applies for reason (b). This allows us to reduce the cycle to a cycle of size 2 (one edge for reason (a), one edge for reason (b)) or to a cycle (size 1) (reason (a)), which corresponds to the fact that there exists a data value x such that $Deq(x)$ happens before $Enq(x)$. Self-loop cycles (size 1) (reason (b)) are not possible, as projections over 1 data value contain only one Deq operation. For cycles of size 2, if d_1 and d_2 are the two data values appearing in the cycle, we have: $h_{|\{d_1,d_2\}} \notin \mathsf{MS}(R_{EnqDeq})$, which is what we wanted to prove. $\square$

Having the small model property, we can build a finite automaton which recognizes each of the small violations, to prove that indeed rule R_{EnqDeq} is co-regular.

Lemma 12. *The rule R_{EnqDeq} is co-regular.*

Proof. We prove in Lemma 11 that if a differentiated history h is not linearizable with respect to $\mathsf{MS}(R_{EnqDeq})$, then it has a projection over 1 or 2 data values which is not linearizable with respect to $\mathsf{MS}(R_{EnqDeq})$ either. Violations of histories with two values are: i) there is a value x such that $Deq(x)$ happens before $Enq(x)$ (or $Enq(x)$ doesn't exist in the history) or ii) there are two operations $Deq(x)$ in h or, iii) there are two values x and y such that $Enq(x)$ happens before $Enq(y)$, and $Deq(y)$ happens before $Deq(x)$ ($Deq(x)$ doesn't exist in the history).



$\mathsf{Ev}_{\mathsf{Queue}}(2)$
$\mathsf{Ev}_{\mathsf{Queue}}(2)$
ret $Deq(1)$
q_8
q_9
call $Deq(1)$
$\mathsf{Ev}_{\mathsf{Queue}}(\{1,2\})$
$\mathsf{Ev}_{\mathsf{Queue}}(\{1,2\})$
$\mathsf{Ev}_{\mathsf{Queue}}(\{1,2\})$
ret $Deq(1)$
ret $Deq(1)$
q_5
q_6
q_7
call $Deq(1)$
$\mathsf{Ev}_{\mathsf{Queue}}(3)$
call $Enq(1)$
ret $Enq(1)$
call $Enq(2)$
ret $Deq(2)$
q_0
q_1
q_2
q_3
q_4
call $Deq(2)$

Figure 2: A non-deterministic automaton recognizing R_{EnqDeq} violations. The top-left branch recognizes executions which have a Deq with no corresponding Enq. The top-right branch recognizes two Deq's returning the same value, which is not supposed to happen in a differentiated execution. The bottom branch recognizes FIFO violations. By the closure properties of implementations, we can assume the `call` $Deq(2)$ are at the beginning.

The automaton $\mathcal{A}_{R_{EnqDeq}}$ in Fig 2 works on the finite alphabet $\mathsf{Ev}_{\mathsf{Queue}}(\{1,2,3\})$ and recognizes all such small violations (top-left branch for i, top-right branch for ii, bottom branch for iii).

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

$$\mathcal{A}_{R_{EnqDeq}} \cap \mathcal{I} \neq \emptyset \iff \exists e \in \mathcal{I}_{\neq}, e' \in \mathsf{proj}(e). \quad \mathtt{last}(e') = R_{EnqDeq} \wedge e' \notin \mathsf{MS}(R_{EnqDeq})$$

($\Rightarrow$) Let $e \in \mathcal{I}$ be an execution which is accepted by $\mathcal{A}_{R_{EnqDeq}}$. By data independence, let $e_{\neq} \in \mathcal{I}$ and r a renaming such that $e = r(e_{\neq})$, and assume without loss of generality that r doesn't rename the data values 1 and 2. If e is accepted by one of the top two branches of $\mathcal{A}_{R_{EnqDeq}}$, we can project $e_{\neq}$ on value 1 to obtain a projection e' such that $\texttt{last}(e') = R_{EnqDeq}$ and $e' \notin \mathsf{MS}(R_{EnqDeq})$. Likewise, if e is accepted by the bottom branch, we can project $e_{\neq}$ on $\{1, 2\}$, and obtain again a projection e' such that $\texttt{last}(e') = R_{EnqDeq}$ and $e' \notin \mathsf{MS}(R_{EnqDeq})$.

($\Leftarrow$) Let $e_{\neq} \in \mathcal{I}_{\neq}$ such that there is a projection e' such that $\texttt{last}(e') = R_{EnqDeq}$ and $e' \notin \mathsf{MS}(R_{EnqDeq})$. As recalled at the beginning of the proof, we know $e_{\neq}$ has to contain a violation of type *i*, *ii*, or *ii*. If it is of type *i* or *ii*, we define the renaming r, which maps x to 1, and all other data values to 2. The execution $r(e_{\neq})$ can then be recognized by on of the top two branches of $\mathcal{A}_{R_{EnqDeq}}$ and belongs to $\mathcal{I}$ by data independence.

Likewise, if it is of type *iii*, r will map x to 1, and y to 2, and all other data values to 3, so that $r(e_{\neq})$ can be recognized by the bottom branch of $\mathcal{A}_{R_{EnqDeq}}$. □

6.2. Co-Regularity of $R_{DeqEmpty}$

As opposed to R_{EnqDeq}, the rule $R_{DeqEmpty}$ doesn't have a small model property. Yet, we show that we can still define a finite automaton to recognize violations. We first define the notion of *covering*[3], which intuitively corresponds to an interval in an execution where the queue cannot be empty.

Definition 9. *Let $h = (O, <, \ell)$ be a differentiated history and $o \in O$. We say that o is* covered *by $d_1, \ldots, d_t \in \mathbb{D}_h$ if*

- *$Enq(d_1)$ happens before o in h, and*
- *$Enq(d_i)$ happens before $Deq(d_{i-1})$ in h for $1 < i$, and*
- *o happens before $Deq(d_t)$, or $Deq(d_t)$ doesn't exist in h.*

The following definitions enable us to compare the times at which two operations end, or the times at which two operations start.

Definition 10. *Let $h = (O, <, \ell)$ be a history, and $o_1, o_2 \in O$. We define the predicate* $\mathsf{endsBefore}(o_1, o_2)$ *that holds if for all $o \in h$, o_2 happens before o implies o_1 happens before o. We say that o_1* ends before *o_2. By definition of interval orders, the relation* endsBefore *is a total order over the operations of h.*

Similarly, we define the predicate $\mathsf{startsBefore}(o_1, o_2)$ *that holds if for all $o \in h$, o happens before o_1 implies o happens before o_2. We say that o_1* starts before *o_2.*

Next, we prove that whenever a history contains a projection that violates rule $R_{DeqEmpty}$, there must exist a *DeqEmpty* operation which is covered.

[3]Our definition is similar to the definition of Henzinger et al. [10].

Lemma 13. *Let h be a differentiated history that has a projection h' with $\texttt{last}(h') = R_{DeqEmpty}$ and $h' \notin \mathsf{MS}(R_{DeqEmpty})$. Then h has a DeqEmpty operation o and data values $d_1, \dots, d_t \in \mathbb{D}_h$ such that o is covered by $d_1, \dots, d_t \in \mathbb{D}_h$.*

Proof. Consider a minimal projection h' of h such that $\texttt{last}(h') = R_{DeqEmpty}$ and $h' \notin \mathsf{MS}(R_{DeqEmpty})$. By definition of $\texttt{last}(h') = R_{DeqEmpty}$, h' contains a *DeqEmpty* operation o.

We sort the *Enq* operations of h' by the $\mathsf{endsBefore}$ comparison operator, and define $d_1, \dots, d_t \in \mathbb{D}_{h'}$ to be the data values corresponding to these *Enq* operations, in order.

We prove that:

1. $Enq(d_1)$ happens before o in h', and
2. $Enq(d_i)$ happens before $Deq(d_{i-1})$ in h' for $1 < i \leq t$, and
3. $Deq(d_{i-1})$ starts before $Deq(d_i)$ in h' for $1 < i < t$, and
4. o happens before $Deq(d_t)$, or $Deq(d_t)$ doesn't exist in h'.

We first prove Condition 1. Assume by contradiction that $Enq(d_1)$ does not happen before o. Then by definition of $\mathsf{endsBefore}$, no *Enq* operation of h ends before o. This entails that $h' \sqsubseteq \mathsf{MS}(R_{DeqEmpty})$, by linearizing o before all *Enq* operations, which contradicts our assumption.

Then, we prove by induction conditions 2 and 3. Assume that for some $1 < j < t$, we have proved that, $Enq(d_i)$ happens before $Deq(d_{i-1})$, and $Deq(d_{i-1})$ starts before $Deq(d_i)$ for all $1 < i \leq j$.

Our goal is first to prove that $Enq(d_{j+1})$ happens before $Deq(d_j)$. Assume by contradiction that $Enq(d_{j+1})$ does not happen before $Deq(d_j)$. Then we can linearize h' into a sequence $u \cdot DeqEmpty \cdot v \in \mathsf{MS}(R_{DeqEmpty})$ where u does not have unmatched *Enq* operations. More precisely, we define u and v so that u contains (at least) all operations with a data value d_i with $1 \leq i \leq j$, and v contains all other enqueue operations in v. This contradicts our assumption that $h' \notin \mathsf{MS}(R_{DeqEmpty})$.

Then if $j < t - 1$, our goal is to prove that $Deq(d_j)$ starts before $Deq(d_{j+1})$. Assume by contradiction that $Deq(d_j)$ does not start before $Deq(d_{j+1})$, i.e. that $Deq(d_{j+1})$ starts before $Deq(d_j)$. Consider the projection h'' of h' that removes the operations with data value d_{j+1}. By minimality of h'', $h'' \sqsubseteq \mathsf{MS}(R_{DeqEmpty})$, and $h'' \sqsubseteq u \cdot DeqEmpty \cdot v$ where u does not have unmatched *Enq* operations. Since $Enq(d_1)$ happens before o, we know $Enq(d_1)$ must belong to u. Since u does not have unmatched *Enq* operations, $Deq(d_1)$ must belong to u. Similarly, using the fact that $Enq(d_i)$ happens before $Deq(d_{i-1})$ for all $1 < i \leq j$, we conclude that $Deq(d_j)$ must belong to u as well. Moreover, since $Enq(d_{j+1})$ happens before $Deq(d_j)$, and $Deq(d_{j+1})$ starts before $Deq(d_j)$, we can define u' such that $h' \sqsubseteq u' \cdot DeqEmpty \cdot v$, and u' contains $Enq(d_{j+1})$, $Deq(d_{j+1})$, and the operations of u. Therefore, u' does not contain unmatched *Enq* operations, which contradicts $h' \notin \mathsf{MS}(R_{DeqEmpty})$.

Finally, we prove Condition 4. Assume by contradiction that $Deq(d_t)$ exists and that o does not happen before $Deq(d_t)$. Following the same reasoning as in the induction above, we can linearize h' into a sequence $u \cdot DeqEmpty \cdot v$, where u does not contain any unmatched Enq operation, by linearizing all operations before o. This contradicts $h' \notin \mathsf{MS}(R_{DeqEmpty})$. □

Conversely, if a $DeqEmpty$ operation is covered, then h must contain a projection which violates rule $R_{DeqEmpty}$.

Lemma 14. *Let h be a differentiated history. If h contains a DeqEmpty operation o and data values $d_1, \dots, d_t \in \mathbb{D}_h$ such that o is covered by $d_1, \dots, d_t \in \mathbb{D}_h$, then h has a projection h' with $\mathtt{last}(h') = R_{DeqEmpty}$ and $h' \notin \mathsf{MS}(R_{DeqEmpty})$.*

Proof. Define h' to be the projection of h that contains operation o, as well as all operations with data values $d_1, \dots, d_t \in \mathbb{D}_h$. Assume by contradiction that $h' \sqsubseteq \mathsf{MS}(R_{DeqEmpty})$. This means there exist u and v such that $h' \sqsubseteq u \cdot DeqEmpty \cdot v$ and u does not contain unmatched Enq operations.

By definition of covering, $Enq(d_1)$ must belong to u. Moreover, since u does not contain unmatched Enq operations, $Deq(d_1)$ must belong to u as well. Since $Enq(d_2)$ happens before $Deq(d_1)$, $Enq(d_2)$ belongs to u too. By continuing this reasoning, we obtain that $Enq(d_t)$ belongs to u as well. Then, we know by the definition of covering that, either $Deq(d_t)$ does not exist, or o happens before $Enq(d_t)$. The first case contradicts that u does not contain unmatched Enq operations, while the second case contradicts the fact that $Deq(d_t)$ must be in u. □

The following corollary is a combination of lemmas 13 and 14, and characterizes the violation with respect to rule $R_{DeqEmpty}$.

Corollary 1 (Characterization of violations with respect to $R_{DeqEmpty}$)**.** *Let h be a differentiated history. There is a projection h' of h such that $\mathtt{last}(h') = R_{DeqEmpty}$ and $h' \notin \mathsf{MS}(R_{DeqEmpty})$ if and only if h contains a DeqEmpty operation o and data values $d_1, \dots, d_t \in \mathbb{D}_h$ such that o is covered by $d_1, \dots, d_t \in \mathbb{D}_h$.*

We now make an automaton to recognize a covering of a $DeqEmpty$ operation, allowing us to detect violations with respect to the rule $R_{DeqEmpty}$.

Lemma 15. *The rule $R_{DeqEmpty}$ is co-regular.*

Proof. We proved in Corollary 1 that a history has a projection such that $\mathtt{last}(h') = R_{DeqEmpty}$ and $h' \notin \mathsf{MS}(R_{DeqEmpty})$ if and only if it has a $DeqEmpty$ operation which is *covered* by other operations, as depicted in Fig 3. The automaton $\mathcal{A}_{R_{DeqEmpty}}$ in Fig 4 recognizes such violations.

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

$$\mathcal{A}_{R_{DeqEmpty}} \cap \mathcal{I} \neq \varnothing \iff \exists e \in \mathcal{I}_{\neq}, e' \in \mathsf{proj}(e). \\ \mathtt{last}(e') = R_{DeqEmpty} \wedge e' \notin \mathsf{MS}(R_{DeqEmpty})$$

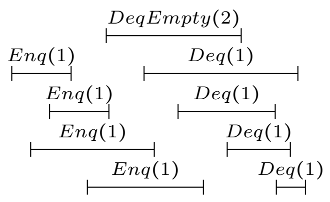


Figure 3: A four-pair $R_{DeqEmpty}$ violation. Lemma 15 demonstrates that this pattern with arbitrarily-many pairs is regular.

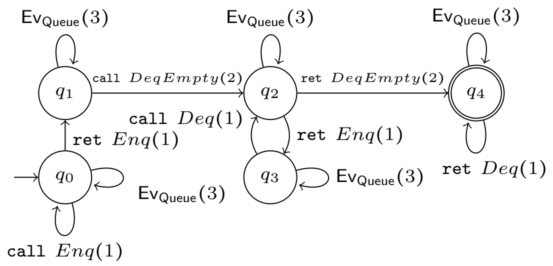

$\mathsf{Ev}_{\mathsf{Queue}}(3)$
$\mathsf{Ev}_{\mathsf{Queue}}(3)$
$\mathsf{Ev}_{\mathsf{Queue}}(3)$
q_1
call $DeqEmpty(2)$
q_2
ret $DeqEmpty(2)$
q_4
call $Deq(1)$
ret $Enq(1)$
ret $Enq(1)$
ret $Deq(1)$
q_0
$\mathsf{Ev}_{\mathsf{Queue}}(3)$
q_3
$\mathsf{Ev}_{\mathsf{Queue}}(3)$
call $Enq(1)$

Figure 4: 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.

($\Rightarrow$) 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 $d_1, \ldots, d_t$ 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 be the *DeqEmpty* operation with data value d. By construction of the automaton we can prove that o is covered by $d_1, \ldots, d_t$, and using Lemma 14, conclude that the history h of e has a projection such that $\mathtt{last}(h') = R_{DeqEmpty}$ and $h' \notin \mathsf{MS}(R_{DeqEmpty})$.

($\Leftarrow$) Let $e_{\neq} \in \mathcal{I}_{\neq}$ such that there is a projection e' such that $\mathtt{last}(e') = R_{DeqEmpty}$ and $e' \notin \mathsf{MS}(R_{DeqEmpty})$. Let $d_1, \ldots, d_t$ be the data values given by Lemma 13, and let d be the data value corresponding to the *DeqEmpty* operation.

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

6.3. *Co-Regularity of the* Stack *rules*

The Stack rule $R_{PushPop}$ is very similar to the $R_{DeqEmpty}$ rule of the Stack. We use the notion of *gap*, which intuitively corresponds to a point in an execution where the Stack could be empty.

Definition 11. *Let $h = (O, <, l)$ be a differentiated history. A* gap *is a partition $O = L \uplus R$ satisfying:*

- *L has no unmatched Push operations, and*
- *no operation of R happens before an operation of L.*

A gap is non-trivial *if $L \neq \emptyset$ and $R \neq \emptyset$.*

Lemma 16. *A differentiated history h has a projection h' such that* $\texttt{last}(h') = R_{PushPop}$ *and* $h' \notin \mathsf{MS}(R_{PushPop})$ *if and only if either:*

- *there exists an unmatched $Pop(d)$ operation, or*
- *there is a $Pop(d)$ which happens before $Push(d)$, or*
- *there exists a projection h'' of h such that:*
  - *for any minimal Push of h'', the corresponding Pop is not maximal in h'', and*
  - *h'' does not have a non-trivial gap.*

Proof. ($\Leftarrow$) We have three cases to consider

- there exists an unmatched $Pop(d)$ operation: define $h' = h_{|\{d\}}$,
- there is a $Pop(d)$ which happens before $Push(d)$: define $h' = h_{|\{d\}}$,
- there exists a projection h'' of h such that:
  - for any minimal Push of h'', the corresponding Pop is not maximal in h'', and
  - h'' does not have a non-trivial gap.

  In that case, we can define h' to be h''.

($\Rightarrow$) Let h' be a projection of h such that $\texttt{last}(h') = R_{PushPop}$ and $h' \notin \mathsf{MS}(R_{PushPop})$. Assume there are no unmatched $Pop(d)$ operations, and that for every d, $Pop(d)$ doesn't happen-before $Push(d)$. This means that h' is made of pairs of $Push(d)$ and $Pop(d)$ operations. We choose h' so that h' is a minimal projection satisfying $\texttt{last}(h') = R_{PushPop}$ and $h' \notin \mathsf{MS}(R_{PushPop})$.

We then define h'' to be h', and prove the necessary conditions on h'. Since $h' \notin \mathsf{MS}(R_{PushPop})$, we know there is no Push operation which is minimal in h' and whose corresponding Pop operation is maximal in h'. It remains to prove that h' does not have a non-trivial gap. Assume by contradiction that h' has a non-trivial gap $L \uplus R$. We choose a non-trivial gap where L is minimal.

By minimality of L, the history $h_L = h'_{|L}$ does not have a non-trivial gap. Moreover, by minimality of h', and h_L being a (strict) projection of h', and since $\texttt{last}(h_L) = R_{PushPop}$, we have $h_L \sqsubseteq \mathsf{MS}(R_{PushPop})$. Therefore, there must exist a minimal $Push(d)$ operation in h_L such that $Pop(d)$ is maximal in h_L. This entails that $h' \sqsubseteq \mathsf{MS}(R_{PushPop})$ (by using d as a witness), contradicting our original assumption.

□

Lemma 17. *The rule $R_{PushPop}$ is co-regular.*

Proof. The automaton Fig 5 recognizes the violations given by Lemma 16. The proof is similar to Lemma 15. The loop over state q_1 ensures there is no non-trivial gap in the execution. □

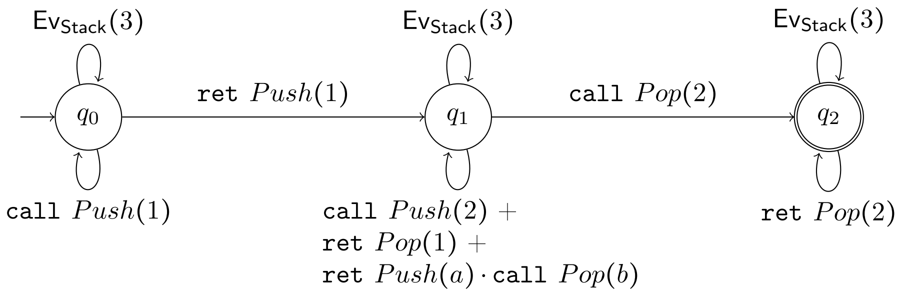

$\mathsf{Ev}_{\mathsf{Stack}}(3)$
$\mathsf{Ev}_{\mathsf{Stack}}(3)$
$\mathsf{Ev}_{\mathsf{Stack}}(3)$
q_0
ret $Push(1)$
q_1
call $Pop(2)$
q_2
call $Push(1)$
call $Push(2)$ + ret $Pop(1)$ + ret $Push(a) \cdot$ call $Pop(b)$
ret $Pop(2)$

Figure 5: An automaton recognizing $R_{PushPop}$ violations. Values a and b range over $\{1, 2\}$.

The rules R_{Push} and $R_{PopEmpty}$ can also be proven co-regular using the same techniques.

Lemma 18. *The rule R_{Push} is co-regular.*

Proof. We can make a characterization of the violations similar to Lemma 16. This rule is in a way simpler, because the *Push* in this rule plays the role of the *Pop* in $R_{PushPop}$. □

Lemma 19. *The rule $R_{PopEmpty}$ is co-regular.*

Proof. Identical to Lemma 15 (replace *Enq* by *Push*, *Deq* by *Pop*, and *DeqEmpty* by *PopEmpty*). □

6.4. Co-Regularity of Register *and* Mutex *rules*

The co-regularity for the rules of the Register and Mutex can be obtained by following the same schema as for the R_{EnqDeq} rule. Here again, we can obtain a small-model property by showing that the minimal cycles in the $\mathbf{W}$ relation are at most of size 2.

7. 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 $\mathcal{C}$ of data-independent implementations for which the latter problem, and thus linearizability, is decidable.

Each method of an implementation in $\mathcal{C}$ 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 $\mathbb{D}$. Computations on data values are forbidden, i.e., the assignments to data variables are only of the form `x := y` where `x` and y are data variables, and data variables cannot be used in boolean conditions (of "if" or "while" statements). This class captures typical implementations, or finite-state abstractions thereof, e.g., obtained via predicate abstraction.

Let $\mathcal{I}$ be an implementation from class $\mathcal{C}$. All the automata $\mathcal{A}$ constructed to prove co-regularity 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 $\mathcal{I}_3 = \{e \in \mathcal{I} \mid e = e_{|\{1,2,3\}}\}$.

We model the set $\mathcal{I}_3$ as the runs of a *Vector Addition System with States (VASS)* similarly to Bouajjani et al. [3]. A VASS has a (finite) set of states, and manipulates a (finite) set of *counters* holding non-negative values, which can be incremented and decremented (but never tested to 0). The fact that the data values provided as arguments of a method are only copied ensures that we only need to represent the values $\{1, 2, 3\}$ inside the implementation.

The idea is that the states of the VASS represent the global variables of $\mathcal{I}$. This set is finite as we have bounded the data values. Each counter of the VASS then represents the number of threads which are at a particular control location within a method, with a certain valuation of the local variables. Here again, as the data values are bounded, there is a finite number of valuations. When a thread moves from a control location to another, or updates its local variables, we decrement, resp., increment, the counter corresponding the old, resp., the new, control location or valuation.

Emptiness of the intersection with finite automata reduces to the EXPSPACE-complete problem of checking reachability in a VASS. Limiting verification to a bounded number of threads lowers the complexity of coverability to PSPACE [6], as this bounds the counters of the VASS. 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 $\mathcal{C}$ 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*[4].

8. Related Work

Several works investigate the theoretical limits of linearizability verification. Verifying a single execution against an arbitrary ADT specification is NP-complete [8]. 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, 9], and undecidable otherwise [3].

Existing automated methods for proving linearizability of an atomic object implementation are also based on reductions to safety verification [1, 10, 12]. Vafeiadis [12] considers implementations where operations' *linearization points*

[4]The size of the implementation is defined as: number of valuations to the shared variables + number of valuations to the local variables + number of control locations (+ number of threads when it is bounded).

are manually specified. 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 [10] 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. [5] 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.

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

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

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

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

[9] J. Hamza. On the complexity of linearizability. In *NETYS '15*. Springer, 2015.

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

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

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