Checking Linearizability of Concurrent Priority Queues[*]

Ahmed Bouajjani[1], Constantin Enea[1], and Chao Wang[1]

1 **IRIF, University Paris Diderot, Paris, France**
`{abou,cenea,wangch}@irif.fr`

Abstract

Efficient implementations of concurrent objects such as atomic collections are essential to modern computing. Unfortunately their correctness criteria — linearizability with respect to given ADT specifications — are hard to verify. Verifying linearizability is undecidable in general, even on classes of implementations where the usual control-state reachability is decidable. In this work we consider concurrent priority queues which are fundamental to many multi-threaded applications like task scheduling or discrete event simulation, and show that verifying linearizability of such implementations is reducible to control-state reachability. This reduction entails the first decidability results for verifying concurrent priority queues with an unbounded number of threads, and it enables the application of existing safety-verification tools for establishing their correctness.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Concurrency, Linearizability, Model Checking

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.12

1 Introduction

Multithreaded software is typically built with specialized "concurrent objects" like atomic integers, queues, maps, priority queues. These objects' methods are designed to confom to better established sequential specifications, a property known as *linearizability* [15], despite being optimized to avoid blocking and exploit parallelism, e.g., by using machine instructions like compare-and-swap. Intuitively, linearizability asks that every individual operation appears to take place instantaneously at some point between its invocation and its return. Verifying linearizability is intrinsically hard, and undecidable in general [4]. However, recent work [5] has shown that for particular objects, e.g., registers, mutexes, queues, and stacks, the problem of verifying linearizability becomes decidable (for finite-state implementations).

In this paper, we consider another important object, namely the priority queue, which is essential for applications such as task scheduling and discrete event simulation. Numerous implementations have been proposed in the research literature, e.g., [2, 8, 17, 20, 19], and concrete implementations exist in many modern languages like C++ or Java. Priority queues are collections providing *put* and *rm* methods for adding and removing values. Every added value is associated to a priority and a remove operation returns a minimal priority value. For generality, we consider a partially-ordered set of priorities. Values with incomparable priorities can be removed in any order, and values having the same priority are removed in the FIFO order. Implementations like the PriorityBlockingQueue in Java where same priority

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

values are removed in an arbitrary order can be modeled in our framework by renaming equal priorities to incomparable priorities (while preserving the order constraints).

Compared to previously studied collections like stacks and queues, the main challenge in dealing with priority queues is that the order in which values are removed is not fixed by the happens-before between add/remove operations (e.g., in the case of queues, values are removed in the order in which they were inserted), but by parameters of the *put* operations (the priorities) which come from an unbounded domain. For instance, the sequential behavior $put(a, p_1) \cdot put(b, p_3) \cdot put(c, p_2) \cdot rm(a, p_1) \cdot rm(c, p_2)$ where the priority p_1 is less than p_2 which is less than p_3, is not admitted neither by the regular queue nor the stack.

We give a characterization of concurrent priority queue behaviors violating linearizability in terms of automata. This characterization enables a reduction of checking linearizability for arbitrary implementations to reachability or invariant checking, and implies decidability for checking linearizability of finite-state implementations. While linearizability violations for stacks and queues can be described using finite-state automata [5], the case of priority queues requires *register automata* where registers are used to store and compare priorities.

This characterization is obtained in several steps. We define a recursive procedure that recognizes valid sequential executions, which is then extended to recognize linearizable concurrent executions. Intuitively, for an execution e, this procedure deals with values occurring in e one by one, starting with values of maximal priority (to be removed the latest). For each value x, it checks whether e satisfies some property “local” to that value, i.e., which is agnostic to how the operations adding or removing other values are ordered between them (w.r.t. the happens-before), other than how they are ordered w.r.t. the operations on x. When this property holds, the procedure is applied recursively on the rest of the execution, without the operations on x. This procedure works only for executions where a value is added at most once, but this is not a limitation for *data-independent* implementations whose behavior doesn’t depend on the values that are added or removed. In fact, all the implementations that we are aware of are data-independent.

Next, we show that checking whether an execution violates this “local” property for a value x can be done using a class of register automata [16, 9, 18] (transition systems where the states consist of a fixed set of registers that can receive values and be compared). Actually, only two registers are needed: one register r_1 for storing a priority guessed at the initial state, and one register r_2 for reading priorities as they occur in the execution and comparing them with the one stored in r_1. We show that registers storing values added to or removed from the priority queue are not needed, since any data-independent implementation admits a violation to linearizability whenever it admits a violation where the number of values is constant, and at most 5 (the number of priorities can still be unbounded).

The remainder of this article is organized as follows. Section 2 describes the priority queue ADT, lists several semantic properties like data-independence, and recalls the notion of linearizability. Section 3 defines a recursive procedure for checking linearizability of concurrent priority queue behaviors. Section 4 gives an automata characterization of the violations to linearizability, and Section 5 discusses related work. Detailed proofs and constructions can be found in the extended version [7].

2 The Priority Queue ADT

We consider priority queues whose interface contains two methods *put* and *rm* for adding and respectively, removing a value. Each value is assigned with a priority when being added to the data structure (by calling *put*) and the remove method *rm* removes a value with a minimal priority. For generality, we assume that the set of priorities is partially-ordered. Incomparable priorities can be removed in any order. When multiple values are assigned with the same priority, *rm* returns the least recent value. Also, when the set of values stored in the priority queue is empty, *rm* returns the distinguished value *empty*. In this section, we formalize (concurrent) executions and implementations, introduce a set of properties satisfied by all the implementations we are aware of, and recall the standard correctness criterion for concurrent implementations of ADTs known as *linearizability* [15].

2.1 Executions

We fix a (possibly infinite) set $\mathbb{D}$ of data values, a (possibly infinite) set $\mathbb{P}$ of priorities, a partial order $\prec$ among elements in $\mathbb{P}$, and an infinite set $\mathbb{O}$ of operation identifiers. The latter are used to match call and return actions of the same invocation. Call actions $call_o(put, a, p)$ and $call_o(rm, a')$ with $a \in \mathbb{D}$, $a' \in \mathbb{D} \cup \{empty\}$, $p \in \mathbb{P}$, and $o \in \mathbb{O}$, combine a method name and a set of arguments with an operation identifier. The return value of a remove is transformed to an argument value for uniformity [1]. The return actions are denoted in a similar way as $ret_o(put, a, p)$ and respectively, $ret_o(rm, a')$.

An *execution* e is a sequence of call and return actions which satisfy the following well-formedness properties: each return is preceded by a matching call (having the same operation identifier), and each operation identifier is used in at most one call/return. We assume every set of executions is closed under isomorphic renaming of operation identifiers. An $m(a)$-operation in an execution e is an operation identifier o s.t. e contains the actions $call_o(m, a)$ and $ret_o(m, a)$. An execution is called *sequential* when no two operations overlap, i.e., each call action is immediately followed by its matching return action, and *concurrent* otherwise. For readability, we write a sequential execution as a sequence of $put(a, p)$ and $rm(a)$ symbols representing a pair of actions $call_o(put, a, p) \cdot ret_o(put, a, p)$ and $call_o(rm, a) \cdot ret_o(rm, a)$, respectively ($o \in \mathbb{O}$). For example, given two priorities $p_1 \prec p_2$, $put(a, p_2) \cdot put(b, p_1) \cdot rm(b)$ is a sequential execution of the priority queue (rm returns b because it has smaller priority).

We define SeqPQ, the set of sequential priority queue executions, semantically via a labelled transition system (LTS, for short). An LTS is a tuple $A = (Q, \Sigma, \to, q_0)$, where Q is a set of states, Σ is an alphabet of transition labels, $\to \subseteq Q \times \Sigma \times Q$ is a transition relation, and q_0 is the initial state. We model the priority queue as an LTS PQ where states are mappings associating priorities in $\mathbb{P}$ with sequences of values in $\mathbb{D}$, representing a snapshot of the priority queue (for each priority, the values are ordered as they were inserted), and the transition labels are $put(a, p)$ and $rm(a)$. Each transition modifies the state as expected. For example, $q_1 \xrightarrow{rm(empty)} q_2$ if $q_1 = q_2$, and q_1 and q_2 map each priority to the empty sequence ϵ. Then, SeqPQ is the set of traces (words) accepted by PQ.

An implementation $\mathcal{I}$ is a set of executions. Implementations represent libraries whose methods are called by external programs. In the remainder of this work, we consider only *completed* executions, where each call action has a corresponding return action. This simplification is sound when the method invocations can always make progress in isolation.

2.2 Semantic Properties of Priority Queues

We define two properties which are important for our results: (1) *data independence* [22, 1] states that priority queue behaviors do not depend on the actual values which are added

[1] Method return values are guessed nondeterministically, and validated at return points. This can be handled using `assume` statements, which only admit executions satisfying a given predicate.

to the queue, and (2) *closure under projection* [5] states that executions remain valid by removing all the operations adding or removing certain values.

An execution e is *data-differentiated* if every value is added at most once, i.e., for each $d \in \mathbb{D}$, e contains at most one action $call_o(put, d, p)$ with $o \in \mathbb{O}$ and $p \in \mathbb{P}$. Note that this property concerns only values, a data-differentiated execution e may contain more than one value with the same priority. The subset of data-differentiated executions of a set of executions E is denoted by $E_{\neq}$.

A renaming function r is a function from $\mathbb{D}$ to $\mathbb{D}$. Given an execution e, we denote by $r(e)$ the execution obtained from e by replacing every data value x by $r(x)$. Note that r renames only the values and keeps the priorities unchanged. Intuitively, renaming values has no influence on the behavior of the priority queue, contrary to renaming priorities.

▶ **Definition 1.** A set of executions E is *data independent* iff
- for all $e \in E$, there exists $e' \in E_{\neq}$ and a renaming function r, such that $e = r(e')$,
- for all $e \in E$ and for all renamings r, $r(e) \in E$.

The following lemma is a direct consequence of definitions.

▶ **Lemma 2.** SeqPQ *is data independent.*

Beyond sequential executions, every concurrent priority queue implementation that we are aware of is data-independent. From now on, we consider only data-independent implementations. This assumption enables a reduction from checking the correctness of an implementation $\mathcal{I}$ to checking the correctness of its data-differentiated executions in $\mathcal{I}_{\neq}$.

Besides data independence, the sequential executions of the priority queue satisfy the following closure property: an execution remains valid when removing all the operations with an argument in some set of values $D \subseteq \mathbb{D}$ and any $rm(empty)$ operation (since they are read-only and they don't affect the queue's state). To distinguish between different $rm(empty)$ operations while simplifying the technical exposition, we assume that they receive as argument a value, i.e., call actions are of the form $call_o(rm, empty, a)$ for some $a \in \mathbb{D}$. We will make explicit this argument only when needed in our technical development. The projection $e|D$ of an execution e to a set of values $D \subseteq \mathbb{D}$ is obtained from e by erasing all the call/return actions with an argument not in D. We write $e \setminus x$ for the projection $e|_{\mathbb{D}\setminus\{x\}}$. Let $proj(e)$ be the set of all projections of e to a set of values $D \subseteq \mathbb{D}$.

▶ **Lemma 3.** SeqPQ *is closed under projection, i.e.,* $proj(e) \subseteq$ SeqPQ *for each* $e \in$ SeqPQ.

2.3 Linearizability

We recall the notion of *linearizability* [15] which is the *de facto* standard correctness condition for concurrent data structures. Given an execution e, the happen-before relation $<_{hb}$ between operations [2] is defined as follows: $o_1 <_{hb} o_2$, if the return action of o_1 occurs before the call action of o_2 in e. The happens-before relation is an interval order [6]: for distinct o_1, o_2, o_3, o_4, if $o_1 <_{hb} o_2$ and $o_3 <_{hb} o_4$, then either $o_1 <_{hb} o_4$, or $o_3 <_{hb} o_2$. Intuitively, this comes from the fact that concurrent threads share a notion of global time.

Given a (concurrent) execution e and a sequential execution s, we say that e is linearizable w.r.t s, denoted $e \sqsubseteq s$, if there is a bijection $f : O_1 \to O_2$, where O_1 and O_2 are the set of operations of e and s, respectively, such that (1) the call and return actions with identifier

[2] In general, we refer to operations using their identifiers.

o and $f(o)$, respectively, are the same and (2) if $o_1 <_{hb} o_2$, then $f(o_1) <_{hb} f(o_2)$. A (concurrent) execution e is linearizable w.r.t. a set S of sequential executions, denoted $e \sqsubseteq S$, if there exists $s \in S$ such that $e \sqsubseteq s$. A set of concurrent executions E is linearizable w.r.t. S, denoted $E \sqsubseteq S$, if $e \sqsubseteq S$ for all $e \in E$.

The following lemma states that by data-independence, it is enough to consider only data-differentiated executions when checking linearizability. Section 3 will focus on characterizing linearizability for data-differentiated executions.

▶ **Lemma 4.** *A data-independent implementation $\mathcal{I}$ is linearizable w.r.t. a data-independent set S of sequential executions, if and only if $\mathcal{I}_{\neq}$ is linearizable w.r.t. $S_{\neq}$.*

3 Checking Linearizability of Priority Queue Executions

We define a recursive procedure for checking linearizability of a data-differentiated execution w.r.t. SeqPQ. To ease the exposition, Section 3.1 introduces a recursive procedure for checking whether a data-differentiated *sequential* execution is admitted by the priority queue which is then extended to the concurrent case in Section 3.2.

3.1 Characterizing Data-Differentiated Sequential Executions

The recursive procedure *Check-PQ-Seq* outlined in Algorithm 1 checks whether a data-differentiated sequential execution belongs to SeqPQ (i.e., if it is accepted by the LTS *PQ*). Roughly, it selects one or two operations in the input execution, checks whether their return values are correct by ignoring the order between the other operations other than how they are ordered w.r.t. the selected ones, and calls itself recursively on the execution without the selected operations.

We explain how the procedure works on the following execution:

$$put(c,p_2) \cdot put(a,p_1) \cdot rm(a) \cdot rm(c) \cdot rm(empty) \cdot put(d,p_2) \cdot put(f,p_3) \cdot rm(f) \cdot put(b,p_1) \quad (1)$$

where p_1, p_2, p_3 are priorities such that $p_1 \prec p_2$ and $p_1 \prec p_3$, and p_2 and p_3 are incomparable. Since the $rm(empty)$ operations are read-only (they don't affect the queue's state), they are selected first. An $rm(empty)$-operation o is correct when every $put(x,p)$ operation before o is matched to a $rm(x)$ operation which also occurs before o. This is true in this case for $x \in \{a,c\}$. Thus, the correctness of (1) reduces to the correctness of

$$put(c,p_2) \cdot put(a,p_1) \cdot rm(a) \cdot rm(c) \cdot put(d,p_2) \cdot put(f,p_3) \cdot rm(f) \cdot put(b,p_1) \quad (2)$$

When the execution contains no $rm(empty)$-operation, the procedure selects a *put* operation adding a value that is not removed and that has a maximal priority. For (2), it selects $put(d,p_2)$ because p_2 is a maximal priority. This operation is correct since d is the last value with priority p_2 in the execution, and the correctness of (2) reduces to the correctness of

$$put(c,p_2) \cdot put(a,p_1) \cdot rm(a) \cdot rm(c) \cdot put(f,p_3) \cdot rm(f) \cdot put(b,p_1) \quad (3)$$

If no operations like above can be found, *Check-PQ-Seq* selects a pair of *put* and *rm* operations adding and removing the same maximal priority value. For (2), it can select $put(c,p_2)$ and $rm(c)$. The value returned by $rm(c)$ is correct if all the values of priority smaller than p_2 added before $rm(c)$ are also removed before $rm(c)$. In this case, a is the only value of priority smaller than p_2 and it satisfies this property. Applying a similar reasoning for all the remaining values, it can be proved that this execution is correct.

Algorithm 1: *Check-PQ-Seq*

```
Input: A data-differentiated sequential execution $e$
Output: true iff $e \in \mathsf{SeqPQ}$
if $e = \epsilon$ then
    return true;
if Has-EmptyRemoves($e$) then
    if $\exists\ o = rm(empty) \in e$ such that EmptyRemove-Seq($e, o$) holds then
        return Check-PQ-Seq($e \setminus o$);
else if Has-UnmatchedMaxPriority($e$) then
    if $\exists\ x \in values(e)$ such that UnmatchedMaxPriority-Seq($e, x$) holds then
        return Check-PQ-Seq($e \setminus x$);
else
    if $\exists\ x \in values(e)$ such that MatchedMaxPriority-Seq($e, x$) holds then
        return Check-PQ-Seq($e \setminus x$);
    else
        return false;
```

Formally, the selected operations depend on the following set of predicates on executions:

$$\mathsf{Has\text{-}EmptyRemoves}(e) = \mathsf{true} \text{ iff } e \text{ contains a } \mathit{rm}(\mathit{empty})\text{-operation}$$

$$\mathsf{Has\text{-}UnmatchedMaxPriority}(e) = \mathsf{true} \text{ iff } p \in \mathit{unmatched\text{-}priorities}(e) \text{ for a maximal } p$$

where *priorities*(e), resp., *unmatched-priorities*(e), is the set of priorities occurring in *put* operations of e, resp., in *put* operations of e for which there is no *rm* operation removing the same value. We call the latter *unmatched* put operations. A put operation which is not unmatched is called *matched*. For simplicity, we consider the following syntactic sugar $\mathsf{Has\text{-}MatchedMaxPriority}(e) = \neg\mathsf{Has\text{-}EmptyRemoves}(e) \wedge \neg\mathsf{Has\text{-}UnmatchedMaxPriority}(e)$. By an abuse of notation, we assume $\mathsf{Has\text{-}UnmatchedMaxPriority}(e) \Rightarrow \neg\mathsf{Has\text{-}EmptyRemoves}(e)$ (this is sound by the order of the conditionals in *Check-PQ-Seq*).

The predicates defining the correctness of the selected operations are defined as follows:

$$\mathsf{EmptyRemove\text{-}Seq}(e, o) = \mathsf{true} \text{ iff } e = u \cdot o \cdot v \text{ and } \mathit{matched}(u)$$

$$\mathsf{UnmatchedMaxPriority\text{-}Seq}(e, x) = \mathsf{true} \text{ iff } e = u \cdot \mathit{put}(x, p) \cdot v,\ p \not\prec \mathit{priorities}(u \cdot v),\ p \notin \mathit{priorities}(v)$$

$$\mathsf{MatchedMaxPriority\text{-}Seq}(e, x) = \mathsf{true} \text{ iff } e = u \cdot \mathit{put}(x, p) \cdot v \cdot \mathit{rm}(x) \cdot w,\ \mathit{matched}_{\prec}(u \cdot v, p),\ p \not\preceq \mathit{unmatched\text{-}priorities}(u \cdot v \cdot w),\ p \not\prec \mathit{priorities}(u \cdot v \cdot w), \text{ and } p \notin \mathit{priorities}(v \cdot w)$$

where $p \prec \mathit{priorities}(e)$ when $p \prec p'$ for some $p' \in \mathit{priorities}(e)$ (and similarly for $p \prec \mathit{unmatched\text{-}priorities}(e)$ or $p \preceq \mathit{unmatched\text{-}priorities}(e)$), $\mathit{matched}_{\prec}(e, p)$ holds when each value with priority strictly smaller than p is removed in e, and *matched*(e) holds when $\mathit{matched}_{\prec}(e, p)$ holds for each $p \in \mathbb{P}$. Compared to the example presented at the beginning of the section, these predicates take into consideration that multiple values with the same priority are removed in FIFO order: the predicate $\mathsf{MatchedMaxPrioritySeq}(e, x)$ holds when x is the last value with priority p added in e.

When o is an *rm*(*empty*)-operation, $e \setminus o$ is the maximal subsequence of e which doesn't contain o. For an execution e, *values*(e) is the set of values in call/return actions of e.

The following lemma states the correctness of *Check-PQ-Seq*.

▶ **Lemma 5.** *Check-PQ-Seq*(e) = true *iff* $e \in \mathsf{SeqPQ}$*, for every data-differentiated sequential execution* e*.*

3.2 Checking Linearizability of Data-Differentiated Concurrent Executions

The extension of *Check-PQ-Seq* to concurrent executions, checking whether they are linearizable w.r.t. SeqPQ, is obtained by replacing every predicate Γ-Seq with

$$\Gamma\text{-}\mathsf{Conc}(e,\alpha) = \mathsf{true} \text{ iff there is a sequential execution } s \text{ such that } e \sqsubseteq s \text{ and } \Gamma\text{-}\mathsf{Seq}(s,\alpha)$$

for each $\Gamma \in \{\mathsf{EmptyRemove}, \mathsf{UnmatchedMaxPriority}, \mathsf{MatchedMaxPriority}\}$. The obtained procedure is denoted by *Check-PQ-Conc* (recursive calls are modified accordingly).

The following lemma states the correctness of *Check-PQ-Conc*. Completeness follows easily from the properties of SeqPQ. If *Check-PQ-Conc*(e) = *false*, then there exists a set D of values s.t. either EmptyRemove-Conc$(e|D)$ is false, or UnmatchedMaxPriority-Conc$(e|D, x)$ is false for all the values x of maximal priority that are not removed (and there exists at least one such value), or MatchedMaxPriority-Conc$(e|D, x)$ is false for all the values x of maximal priority (and these values are all removed in $e|D$). It can be easily seen that we get $e|D \not\sqsubseteq$ SeqPQ in all cases, which by the closure under projection of SeqPQ implies, $e \not\sqsubseteq$ SeqPQ (since every linearization of e includes as a subsequence a linearization of $e|D$).

▶ **Lemma 6.** *Check-PQ-Conc*(e) = true *iff* $e \sqsubseteq$ SeqPQ*, for every data-differentiated* e*.*

Proving soundness is highly non-trivial and one of the main technical contributions of this paper. The main technical difficulty is showing that for any execution e, any linearization of $e \setminus x$ for some maximal priority value x can be extended to a linearization of e provided that UnmatchedMaxPriority or MatchedMaxPriority holds (depending on whether there are values with the same priority as x in e which are not removed).

We explain the proof of this property on the execution e in Figure 1(a) where $p_1 \prec p$, $p_1 \prec p_2$, and the predicate Has-MatchedMaxPriority(e) holds. Assume that there exist two sequential executions l and l' such that $e \sqsubseteq l = u \cdot put(x,p) \cdot v \cdot rm(x) \cdot w$, MatchedMaxPriority-Seq$(l, x)$ holds, and $e \setminus x \sqsubseteq l' \in$ SeqPQ. Let $u = \epsilon$, w be any sequence formed of $put(z_2, p_2)$ and $rm(z_1)$ (we distinguish them by adding the suffix $-w$ to their name, e.g., $rm(z_1) - w$), and v be any sequence containing the remaining operations. In general, the linearization l' can be defined by choosing for each operation, a point in time between its call and return actions, called *linearization point*. The order between the linearization points defines the sequence l'. Figure 1(a) draws linearization points for the operations in $e \setminus x$ defining the linearization l' [3]. We show how to construct a sequence $l'' = l''_1 \cdot put(x,p) \cdot l''_2 \cdot rm(x) \cdot l''_3 \in$ SeqPQ such that $e \sqsubseteq l''$.

- An operation is called p-comparable (resp., p-incomparable) when it receives as argument a value of priority comparable to p (resp., incomparable to p). We could try to define l''_1, l''_2 and l''_3 as the projection of l' to the operations in u, v and w, respectively. However, this is incorrect, since MatchedMaxPriority-Seq(l, x) imposes no restriction on p-incomparable operations in $u \cdot v$, and thus, there is no guarantee that the projection of l' to p-incomparable operations in $u \cdot v$ is correct. In this example, this projection is $put(z_1, p_2) \cdot rm(z_2)$ which is incorrect.
- We define the sets of operations U', V' and W' such that l''_1, l''_2 and l''_3 are the projections of l' to U', V', and W', respectively. This is done in two steps:

[3] In general, there may exist multiple ways of choosing linearization points to define the same linearization. Our construction is agnostic to this choice.

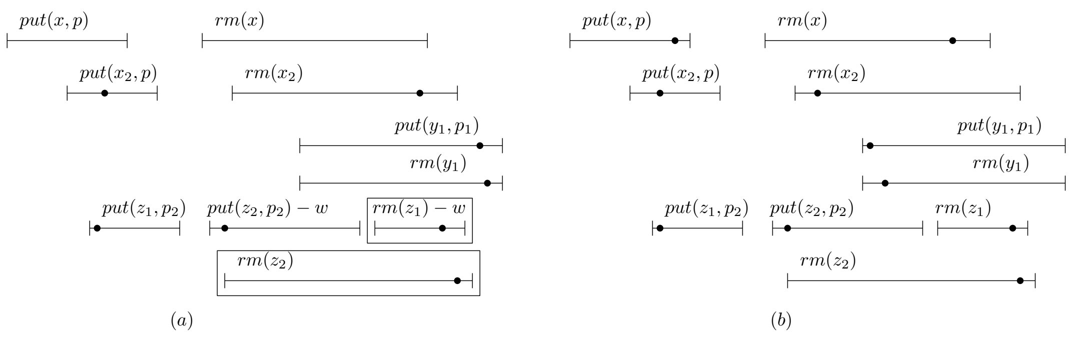

Figure 1 The process of obtaining linearization of e

- The first step is to define W'. The p-comparable operations in W' are the same as in w. To identify the p-incomparable operations in W', we search for a p-incomparable operation o which either happens before some p-comparable operation in w, or whose linearization point occurs after $ret(rm, x)$. We add to W' the operation o and all the p-incomparable operations occurring after o in l'. In this example, o is $rm(z_1)$ and the only p-incomparable operation occurring after o in l' is $rm(z_2)$ (they are surrounded by boxes in the figure). In this process, whether a p-incomparable operation is in W' or not only relies on whether it is before or after such an o in l'.
- The second step is to define U' and V'. U' contains two kinds of operations: (1) operations whose linearization points are before $ret(put, x, p)$, and (2) other put operations with priority p. V' contains the remaining operations. In this example, U' contains $put(z_1, p_2)$ and $put(x_2, p)$.
- In conclusion, we have that $l''_1 = put(z_1, p_2) \cdot put(x_2, p)$, $l''_2 = put(z_2, p_2) \cdot rm(x_2) \cdot put(y_1, p_1) \cdot rm(y_1)$, and $l''_3 = rm(z_1) \cdot rm(z_2)$. Figure 1(b) draws linearization points for each operation in e defining the linearization l''.

Section 4 introduces a characterization of concurrent priority queue violations using a set of *non-recursive* automata (i.e., whose states consist of a fixed number of registers) whose standard synchronized product is equivalent to *Check-PQ-Conc* (modulo renaming of values which is possible by data-independence). Since SeqPQ is closed under projection (Lemma 3), the recursion in *Check-PQ-Conc* can be eliminated by checking that each projection of a given execution e passes a non-recursive version of *Check-PQ-Conc* where every recursive call is replaced by true. More precisely, every occurrence of **return** *Check-PQ-Conc* is replaced by **return** true. Let *Check-PQ-Conc-NonRec* be the thus obtained procedure.

▶ **Lemma 7.** *Given a data-differentiated execution e, $e \sqsubseteq$ SeqPQ if and only if for each $e' \in proj(e)$, Check-PQ-Conc-NonRec(e') returns* true.

4 Reducing Linearizability of Priority Queues to Reachability

We show that the set of executions for which some projection fails the test *Check-PQ-Conc-NonRec* can be characterized using a set of register automata, modulo a value renaming. The possibility of renaming values (which is complete for checking data independent implementations) allows to simplify the reasoning about projections. Thus, we assume that all the operations which are not in the projection failing this test use the same distinguished value $\top$, different from those used in the projection. Then, it is enough to find an automata characterization for the set of executions e for which *Check-PQ-Conc-NonRec* fails, or equivalently, for which one of the following three formulas is false:

$$\Gamma(e) := \mathsf{Has}\text{-}\Gamma(e) \Rightarrow \exists \alpha.\ \Gamma\text{-}\mathsf{Conc}(e, \alpha) \text{ with } \Gamma \in \{\mathsf{EmptyRemove}, \mathsf{UnmatchedMaxPriority}, \mathsf{MatchedMaxPriority}\}$$

Intuitively, $\Gamma(e)$ states that e is linearizable w.r.t. the set of sequential executions described by Γ-Seq (provided that Has-$\Gamma(e)$ holds). Therefore, by an abuse of terminology, an execution e satisfying $\Gamma(e)$ is called *linearizable w.r.t.* Γ, or Γ-*linearizable*. Extending the automaton characterizing executions which are not Γ-linearizable, with self-loops that allow any operation with parameter $\top$ results in an automaton satisfying the following property called Γ-*completeness*.

▶ **Definition 8.** For $\Gamma \in \{\mathsf{EmptyRemove}, \mathsf{UnmatchedMaxPriority}, \mathsf{MatchedMaxPriority}\}$, an automaton A is called Γ-*complete* when for each data-independent implementation $\mathcal{I}$:
$A \cap \mathcal{I} \neq \emptyset$ if and only if there exists $e \in \mathcal{I}$ and $e' \in \mathit{proj}(e)$ such that e' is not Γ-linearizable.

Section 4.1 describes a MatchedMaxPriority-complete automaton, the other automata can be found in the long version [7]. Therefore, the following holds.

▶ **Lemma 9.** *There exists a* Γ-*complete automaton for each* $\Gamma \in \{\mathsf{EmptyRemove}, \mathsf{UnmatchedMaxPriority}, \mathsf{MatchedMaxPriority}\}$.

When defining Γ-complete automata, we assume that every implementation $\mathcal{I}$ behaves correctly, i.e., as a FIFO queue, when only values with the same priority are observed. More precisely, we assume that for every execution $e \in \mathcal{I}$ and every priority $p \in \mathbb{P}$, the projection of e to values with priority p is linearizable (w.r.t. SeqPQ). This property can be checked separately using the techniques introduced for regular FIFO queues in [5] This property can be checked separately using register automata (can be found in the long version [7]) obtained from the finite automata in [5] for FIFO queue. Note that this assumption excludes some obvious violations, such as a $rm(a)$ operation happens before a $put(a, p)$ operation, for some p.

Also, we consider Γ-complete automata for $\Gamma \in \{\mathsf{UnmatchedMaxPriority}, \mathsf{MatchedMaxPriority}\}$, recognizing executions which contain only one maximal priority. This is possible because any data-differentiated execution for which $\Gamma(e)$ is false has such a projection. Formally, given a data-differentiated execution e and p a maximal priority in e, $e|_{\preceq p}$ is the projection of e to the set of values with priorities smaller than p. Then,

▶ **Lemma 10.** *Let* $\Gamma \in \{\mathsf{UnmatchedMaxPriority}, \mathsf{MatchedMaxPriority}\}$ *and* e *a data-differentiated execution. Then,* e *is* Γ-*linearizable iff* $e|_{\preceq p}$ *is* Γ-*linearizable for some maximal priority* p *in* e.

Proof. (Sketch) To prove the *only if* direction, let e be a data-differentiated execution linearizable w.r.t. $l = u \cdot put(x, p) \cdot v \cdot rm(x) \cdot w \in \mathsf{MatchedMaxPriority}\text{-}\mathsf{Seq}(s, x)$. Since $\mathsf{MatchedMaxPriority}\text{-}\mathsf{Seq}(s, x)$ imposes no restriction on the operations in u, v and w with priorities incomparable to p, erasing all these operations results in a sequential execution which still satisfies this property. Similarly, for $\Gamma = \mathsf{UnmatchedMaxPriority}$.

The *if* direction follows from the fact that if the projection of an execution to a set of operations O_1 has a linearization l_1 and the projection of the same execution to the remaining set of operations has a linearization l_2, then the execution has a linearization which is defined as an interleaving of l_1 and l_2.

Thus, let e be an execution such that $e|_{\preceq p}$ is linearizable w.r.t. $l = u \cdot put(x, p) \cdot v \cdot rm(x) \cdot w \in \mathsf{MatchedMaxPriority}\text{-}\mathsf{Seq}(s, x)$. By the property above, we know that e has a linearization $l' = u' \cdot put(x, p) \cdot v' \cdot rm(x) \cdot w'$, such that the projection of l' to values of priority comparable to p is l. Since $\mathsf{MatchedMaxPriority\text{-}Seq}(s, x)$ does not have a condition on values of priority incomparable to p, we obtain that $l' \in \mathsf{MatchedMaxPriority\text{-}Seq}(s, \alpha)$. ◀

The following shows that Γ-complete automata enable an effective reduction of checking linearizability of concurrent priority queue implementations to state reachability. It is a direct consequence of the above definitions. Section 4.2 discusses decidability results implied by this reduction.

▶ **Theorem 11.** *Let $\mathcal{I}$ be a data-independent implementation, and $A(\Gamma)$ be a Γ-complete automaton for each Γ. Then, $\mathcal{I} \sqsubseteq \mathsf{SeqPQ}$ if and only if $\mathcal{I} \cap A(\Gamma) = \emptyset$ for all Γ.*

4.1 A MatchedMaxPriority-complete automaton

A differentiated execution e is not MatchedMaxPriority-linearizable when all the *put* operations in e using the maximal priority p are matched, and e is not linearizable w.r.t. the set of sequential executions satisfying $\mathsf{MatchedMaxPriority\text{-}Seq}(e, x)$ for each value x of priority p. We consider two cases depending on whether e contains exactly one value with priority p or at least two values. We denote by $\mathsf{MatchedMaxPriority}^{>}$ the strengthening of MatchedMaxPriority with the condition that all the values other than x have a priority strictly smaller than p (corresponding to the first case), and by $\mathsf{MatchedMaxPriority}^{=}$ the strengthening of the same formula with the negation of this condition (corresponding to the second case). We use particular instances of register automata [16, 9, 18] whose states include only two registers, one for storing a priority guessed at the initial state, and one for storing the priority of the current action in the execution. The transitions can check equality or the order relation $\prec$ between the values stored in the two registers. Instead of formalizing the full class of register automata, we consider a simpler class which suffices our needs. More precisely, we consider a class of labeled transition systems whose states consist of a finite control part and a register r interpreted to elements of $\mathbb{P}$. The transition labels can be one of the following:

- $r = *$ for storing an arbitrary value to r,
- $call(rm, a)$ and $ret(rm, a)$ for reading call/return actions of a remove,
- $call(put, d, g)$ where $g \in \{= r, \prec r, true\}$ is a guard, for reading a call action $call(put, d, p)$ of a put and checking whether p is either equal to or smaller than the value stored in r, or arbitrary,
- $ret(put, d, true)$ for reading a return action $ret(put, d, p)$ for any p.

The set of words accepted by such a transition system can be defined as usual.

4.1.1 A $\mathsf{MatchedMaxPriority}^{>}$-complete automaton

We give a typical example of an execution e which is not $\mathsf{MatchedMaxPriority}^{>}$-linearizable in Figure 2. Intuitively, this is a violation because during the whole execution of $rm(b)$, the priority queue stores a smaller priority value (which should be removed before b). To be more precise, we define *the interval of a value x* as the time interval from the return of a put $ret(put, x, p)$ to the call of the matching remove $call(rm, x)$, or to the end of the execution if such a call action doesn't exist. Intuitively, it represents the time interval in which a value is guaranteed to be stored into the concurrent priority queue. Concretely, for a standard indexing of actions in an execution, a time interval is a closed interval between the indexes of two actions in the execution. In Figure 2, we draw the interval of each value by dashed line. Here we assume that $p_1 \prec p_4$, $p_2 \prec p_4$, and $p_3 \prec p_4$. We can not find a sequence s where $e \sqsubseteq s$ and $\mathsf{MatchedMaxPriority\text{-}Seq}(s, b)$ holds, since each time point from $call(rm, b)$

to $ret(rm, b)$ is included in the interval of some smaller priority value, and $rm(b)$ can't take effect in the interval of a smaller priority value.

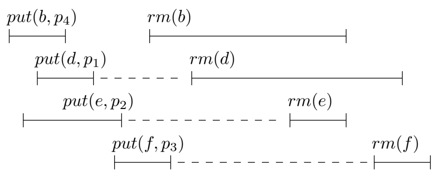

Figure 2 An execution that is not MatchedMaxPriority$^{>}$-linearizable. We represent each operation as a time interval whose left, resp., right, bound corresponds to the call, resp., return action. Operations adding and removing the same value are aligned vertically.

To formalize the scenario in Figure 2 we use the notion of *left-right constraint* defined below.

▶ **Definition 12.** Let e be a data-differentiated execution which contains only one maximal priority p, and only one value x of priority p (and no $rm(empty)$ operations). The *left-right constraint of x* is the graph G where:

- the nodes are the values occurring in e,
- there is an edge from d_1 to x, if $put(d_1, \_) <_{hb} put(x, p)$ or $put(d_1, \_) <_{hb} rm(x)$,
- there is an edge from x to d_1, if $rm(x) <_{hb} rm(d_1)$ or $rm(d_1)$ does not exists,
- there is an edge from d_1 to d_2, if $put(d_1, \_) <_{hb} rm(d_2, \_)$.

The execution in Figure 2 is not MatchedMaxPriority$^{>}$-linearizable because the left-right constraint of the maximal priority value b contains a cycle: $f \to e \to d \to b \to f$. The presence of such a cycle is equivalent to the execution not being MatchedMaxPriority$^{>}$-linearizable, as indicated by the following lemma:

▶ **Lemma 13.** *Given a data-differentiated execution e where* Has-MatchedMaxPriority(e) *holds, let p be its maximal priority and $put(x, p), rm(x)$ are only operations of priority p in e. Let G be the graph representing the left-right constraint of x. e is* MatchedMaxPriority-*linearizable, if and only if G has no cycle going through x.*

When the left-right constraint of the maximal priority value x contains a cycle of the form $d_1 \to \ldots \to d_m \to x \to d_1$ for some $d_1,\ldots,d_n \in \mathbb{D}$, we say that x is *covered* by $d_1, \ldots, d_m$. The shape of such a cycle (i.e., the alternation between call/return actions of put/rm operations) can be detected using our class of automata, the only complication being the unbounded number of values $d_1,\ldots,d_n$. However, by data independence, whenever an implementation contains such an execution it also contains an execution where all the values $d_1,\ldots,d_n$ are renamed to the same value a, and x is renamed to b. Therefore, our automata can be defined over a fixed set of values a, b, and $\top$ (recall that $\top$ is used for the operations outside of the non-linearizable projection).

To define a MatchedMaxPriority$^{>}$-complete automaton we need to consider several cases depending on the order between the call/return actions of the put/rm operations that add and respectively, remove the value b. For example, the automata for the case where the put happens-before the remove (as in Figure 2) is pictured in Figure 3. This automaton captures the three possible ways of ordering the first action $ret(put, a, \_)$ w.r.t. the actions with value b, which are pictured in Figure 4 (a) (this action cannot occur after $call(rm, b, \_)$ since b must be covered by the a-s). The paths corresponding to these three possible orders are: $q_1 \to q_2 \to q_3 \ldots \to q_7$, $q_1 \to q_2 \to q_{10} \ldots \to q_7$ and $q_1 \to q_9 \to q_{10} \ldots \to q_7$. In

Figure 4, we show all the four orders of the call/return actions of adding and removing b, and also possible orders of the first $ret(put, a, \_)$ w.r.t the actions with value b. In the long version [7], three register automata is constructed according to the cases of Figure 4 (b), (c) and (d), respectively. Automata for the other cases can be found in Appendix ??.

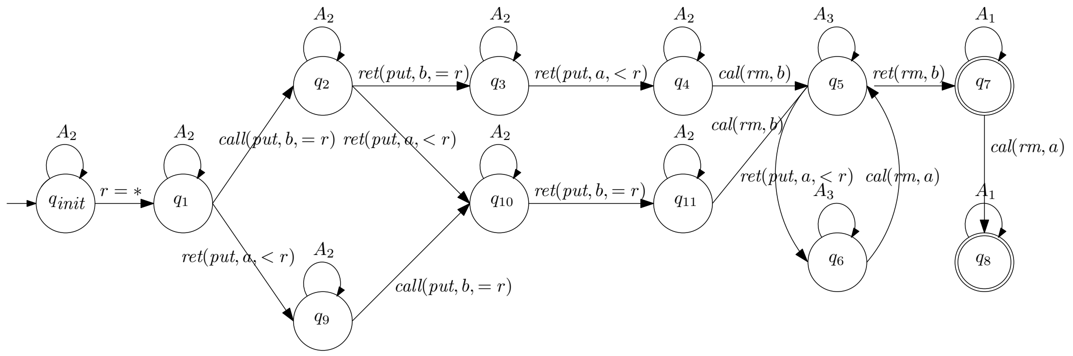

Figure 3 Register automaton $\mathcal{A}^1_{l\text{-}lar}$. We use the following notations: $A_1 = A \cup \{ret(rm, a)\}$, $A_2 = A \cup \{call(put, a, = r)\}$, $A_3 = A_2 \cup \{ret(rm, a)\}$, where $A = \{call(put, \top, true), ret(put, \top, true), call(rm, d), ret(rm, d), call(rm, empty), ret(rm, empty)\}$.

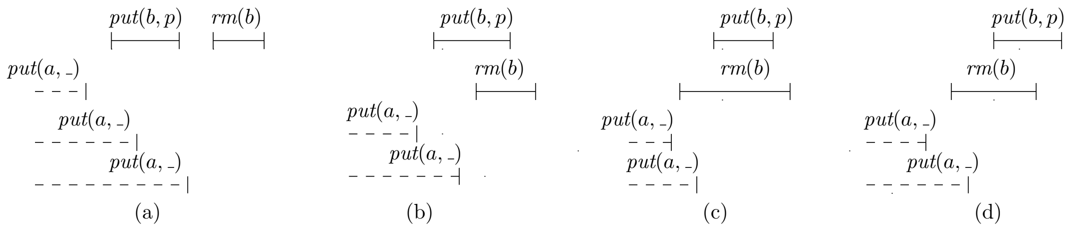

Figure 4 Four cases of ordering actions with value b, and possible orders of the first $ret(put, a, \_)$ w.r.t the actions with value b

4.1.2 A MatchedMaxPriority$^=$-complete automaton

When an execution contains at least two values of maximal priority, the acyclicity of the left-right constraints (for all the maximal priority values) is not enough to conclude that the execution is MatchedMaxPriority-linearizable. Intuitively, there may exist a value a which is added before another value b such that all the possible linearization points of $rm(b)$ are disabled by the position of $rm(a)$ in the happens-before. We give an example of such an execution e in Figure 5. This execution is not linearizable w.r.t. MatchedMaxPriority (or MatchedMaxPriority$^=$) even if neither a nor b are covered by values with smaller priority. Since $put(a, p_4) <_{hb} put(b, p_4)$ and values of the same priority are removed in FIFO order, $rm(a)$ should be linearized before $rm(b)$ (i.e., this execution should be linearizable w.r.t. a sequence where $rm(a)$ occurs before $rm(b)$). Since $rm(b)$ cannot take effect during the interval of a smaller priority value, it could be linearized only in one of the two time intervals pictured with dotted lines in Figure 5. However, each of these time intervals ends before $call(rm, a)$, and thus $rm(a)$ cannot be linearized before $rm(b)$.

To recognize the scenarios in Figure 5, we introduce an order $<_{pb}$ between values which intuitively, can be thought of as "a value a is put before another value b". Thus, given a data-differentiated execution e and two values a and b of maximal priority, $a <_{pb} b$ if one of the

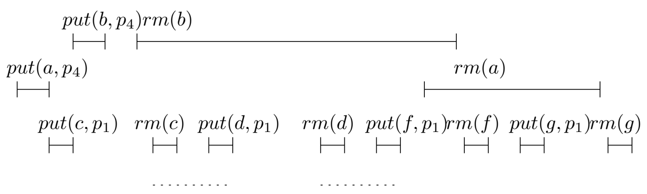
$put(b,p_4)rm(b)$
$put(a,p_4)$
$rm(a)$
$put(c,p_1)$ $rm(c)$ $put(d,p_1)$ $rm(d)$ $put(f,p_1)rm(f)$ $put(g,p_1)rm(g)$

Figure 5 An execution that is not linearizable w.r.t $\mathsf{MatchedMaxPriority}^=$.

following holds: (1) $put(a,\_) <_{hb} put(b,\_)$, (2) $rm(a) <_{hb} rm(b)$, or (3) $rm(a) <_{hb} put(b,\_)$. Sometimes we use $a <^A_{pb} b$, $a <^B_{pb} b$ and $a <^C_{pb} b$ to explicitly distinguish between these three cases. Let $<^*_{pb}$ be the transitive closure of $<_{pb}$.

Then, to model the time intervals in which a remove operation like $rm(b)$ in Figure 5, can be linearized (outside of intervals of smaller priority values) we introduce the notion of gap-point. This notion relies on a standard indexing of actions in an execution, starting with 0. Here we assume that the index of actions of an execution starts from 0.

▶ **Definition 14.** Let e be a data-differentiated execution which contains only one maximal priority p, and $put(x,p)$ and $rm(x)$ two operations in e. An index $i \in [0, |e|-1]$ is a *gap-point of x* if i is greater than or equal to the index of both $call(put,x,p)$ and $call(rm,x)$, smaller than the index of $ret(rm,x)$, and it is not included in the interval of some value with priority smaller than p.

The case of Figure 5 can be formally described as follows: $a <^*_{pb} b$ while the right-most gap-point of b is before $call(rm,a)$ or $call(put,a,p_4)$. The following lemma states that these conditions are enough to characterize non-linearizability w.r.t $\mathsf{MatchedMaxPriority}^=$.

▶ **Lemma 15.** *Let e be a data-differentiated execution which contains only one maximal priority p such that* $\mathsf{Has\text{-}MatchedMaxPriority}(e)$ *holds. Then, e is not linearizable w.r.t* $\mathsf{MatchedMaxPriority}^=$ *iff e contains two values x and y of maximal priority p such that $y <^*_{pb} x$, and the rightmost gap-point of x is strictly smaller than the index of $call(put,y,p)$ or $call(rm,y)$.*

To characterize violations to linearizability w.r.t. $\mathsf{MatchedMaxPriority}^=$ using an automaton that tracks a bounded number of values, we show that the number of values needed to witness that $y <^*_{pb} x$ for some x and y is bounded.

▶ **Lemma 16.** *Let e be a data-differentiated execution such that $a <_{pb} a_1 <_{pb} \ldots <_{pb} a_m <_{pb} b$ holds for some set of values a, a_1,. . .,a_m, b. Then, one of the following holds:*
- *$a <^A_{pb} b$, $a <^B_{pb} b$, or $a <^C_{pb} b$,*
- *$a <^A_{pb} a_i <^B_{pb} b$ or $a <^B_{pb} a_i <^A_{pb} b$, for some i.*

To characterize violations to linearizability w.r.t. $\mathsf{MatchedMaxPriority}^=$, one has to consider all the possible orders between call/return actions of the operations on values a, b, and a_i in Lemma 16, and the right-most gap point of b. Excluding the inconsistent cases, we are left with 5 possible orders that are listed in Appendix **??**we are left with 5 possible orders that are shown in Figure 6, where o denotes the rightmost gap-point of b. For each case, we define an automaton recognizing the induced set of violations. For instance, one such case and the corresponding register automaton is shown in Figure 7 (o denotes the rightmost gap-point of b). For instance, the register automata for the case of Figure 6(a) is shown in Figure 7. In this case, the conditions in Lemma 15 are equivalent to the fact that intuitively, the time interval from $call(rm,a)$ to $ret(rm,b)$ is covered by lower priority values (and thus, there is no gap-point of b which occurs after $call(rm, a)$). Using again the data-independence property, these lower priority values can all be renamed to a fixed value d, and the other values to a fixed value $\top$. We show how to generate register automata for other cases in the long version [7].

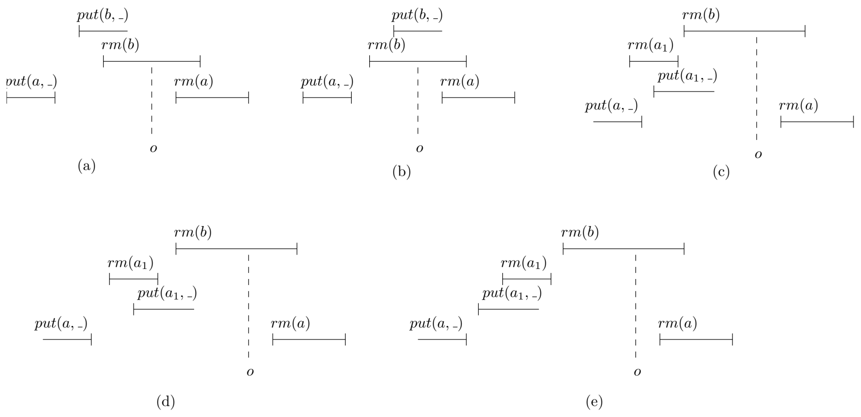

Figure 6 Five cases that need to be considered.

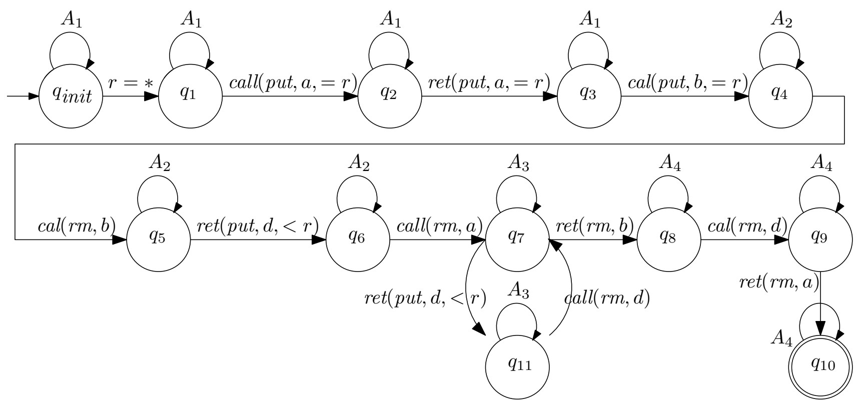

Figure 7 A case in deriving a MatchedMaxPriority$^{=}$-complete automaton. We use the following notations: $C = \{call(put, \top, true), ret(put, \top, true), call(rm, \top), ret(rm, \top), call(rm, empty), ret(rm, empty)\}$, $C_1 = C \cup \{call(put, d, < r)\}$, $C_2 = C_1 \cup \{ret(put, b, = r)\}$, $C_3 = C_2 \cup \{ret(rm, d)\}$, $C_4 = C \cup \{ret(put, b, = r), ret(rm, d)\}$.

4.2 Decidability Result

We describe a class $\mathcal{C}$ of data-independent implementations for which linearizability w.r.t. SeqPQ is decidable. The implementations in $\mathcal{C}$ allow an unbounded number of values but a bounded number of priorities. Each method 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-then-else 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. Since the Γ-complete automata $A(\Gamma)$ we define useSince the Γ-complete automata $A(\Gamma)$ uses a fixed set $D = \{a, b, a_1, d, e, \top\}$ of values, we have that $\mathcal{C} \cap A(\Gamma) \neq \emptyset$ for some Γ iff

$\mathcal{C}_D \cap A(\Gamma) \neq \emptyset$ where $\mathcal{C}_D$ is the subset of $\mathcal{C}$ that uses only values in D. The set of executions $\mathcal{C}_D$ can be represented by a Petri Net since bounding the set of values allows us to represent each thread with a finite-state machine. For a fixed set of priorities P, the register automata $A(\Gamma)$ can be transformed to finite-state automata since the number of possible valuations of the registers is bounded.

The set of executions $\mathcal{C}_D$ can be represented by a Vector Addition Systems with States (VASS), since both values and priorities are finite, which implies that each thread and register automata can be transformed into finite-state automata. To obtain this transformation, the states of the VASS represent the global variables of $\mathcal{C}_D$, while 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. In this way we transform linearizability problem of priority queue into state-reachability problem of VASS, which is EXPSPACE-complete. When we consider only finite number of threads, the value of counters are bounded and then this problem is PSPACE [11].

On the other hand, we can mimic transitions of VASS by a priority queue implementation. To obtain this, we keep *rm* unchanged, while in each *put* method, we first put a value into priority queue, and then either simulate one step of transitions or record one increase/decrease of a counter, similarly as that in [4]. In this way we transform the state-reachability problem of VASS into linearizability problem of priority queue. Based on above discussion, we have the following complexity result.

▶ **Theorem 17.** *Verifying whether an implementation in $\mathcal{C}$ is linearizable w.r.t.* SeqPQ *is PSPACE-complete for a fixed number of threads, and EXPSPACE-complete otherwise.*

5 Related work

The theoretical limits of checking linearizability have been investigated in previous works. Checking linearizability of a single execution w.r.t. an arbitrary ADT is NP-complete [12] while checking linearizability of all the executions of a finite-state implementation w.r.t. an arbitrary ADT specification (given as a regular language) is EXPSPACE-complete when the number of program threads is bounded [3, 13], and undecidable otherwise [4].

Existing automated methods for proving linearizability of a concurrent object implementation are also based on reductions to safety verification, e.g., [1, 14, 21]. The approach in [21] considers implementations where operations' *linearization points* 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 [14] 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. The work in [10] proves 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.

Our previous work [5] shows that checking linearizability of finite-state implementations of concurrent queues and stacks is decidable. Roughly, we follow the same schema: the recursive procedure in Section 3.1 is similar to the inductive rules in [5], and its extension to concurrent executions in Section 3.2 corresponds to the notion of step-by-step linearizability in [5]. Although similar in nature, defining these procedures and establishing their correctness require proof techniques which are specific to the priority queue semantics. The order in which values are removed from a priority queue is encoded in their priorities which come from an unbounded domain, and not in the happens-before order as in the case of stacks and queues. Therefore, the results we introduce in this paper cannot be inferred from those in [5]. At a technical level, characterizing the priority queue violations requires a more expressive class of automata (with registers) than the finite-state automata in [5].

References

1 Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS*, pages 324–338, 2013.

2 Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: a scalable relaxed priority queue. In Albert Cohen and David Grove, editors, *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 11–20. ACM, 2015. URL: http://doi.acm.org/10.1145/2688500.2688523, doi:10.1145/2688500.2688523.

3 Rajeev Alur, Kenneth L. McMillan, and Doron Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000.

4 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Verifying concurrent programs against sequential specifications. In *ESOP '13*, volume 7792 of *LNCS*, pages 290–309. Springer, 2013.

5 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. On reducing linearizability to state reachability. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 2015. URL: http://dx.doi.org/10.1007/978-3-662-47666-6_8, doi:10.1007/978-3-662-47666-6_8.

6 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Tractable refinement checking for concurrent objects. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 651–662. ACM, 2015. URL: http://doi.acm.org/10.1145/2676726.2677002, doi:10.1145/2676726.2677002.

7 Ahmed Bouajjani, Constantin Enea, and Chao Wang. Checking linearizability of concurrent priority queues. *CoRR*, 2017. URL: https://arxiv.org/abs/1707.00639.

8 Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. The adaptive priority queue with elimination and combining. In Fabian Kuhn, editor, *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, volume 8784 of *Lecture Notes in Computer Science*, pages 406–420. Springer, 2014. URL: http://dx.doi.org/10.1007/978-3-662-45174-8_28, doi:10.1007/978-3-662-45174-8_28.

9 Karlis Cerans. Deciding properties of integral relational automata. In Serge Abiteboul and Eli Shamir, editors, *Automata, Languages and Programming, 21st International Colloquium, ICALP94, Jerusalem, Israel, July 11-14, 1994, Proceedings*, volume 820 of *Lecture Notes in Computer Science*, pages 35–46. Springer, 1994. URL: http://dx.doi.org/10.1007/3-540-58201-0_56, doi:10.1007/3-540-58201-0_56.

10 Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack. In *POPL '15*. ACM, 2015.

11 Javier Esparza. Decidability and complexity of petri net problems - an introduction. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 374–428. Springer, 1996.

12 Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM J. Comput.*, 26(4):1208–1244, 1997.

13 Jad Hamza. On the complexity of linearizability. In *NETYS '15*. Springer, 2015.

14 Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR*, pages 242–256, 2013.

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

16 Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994. URL: `http://dx.doi.org/10.1016/0304-3975(94)90242-9`, `doi:10.1016/0304-3975(94)90242-9`.

17 Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In Roberto Baldoni, Nicolas Nisse, and Maarten van Steen, editors, *Principles of Distributed Systems - 17th International Conference, OPODIS 2013, Nice, France, December 16-18, 2013. Proceedings*, volume 8304 of *Lecture Notes in Computer Science*, pages 206–220. Springer, 2013. URL: `http://dx.doi.org/10.1007/978-3-319-03850-6_15`, `doi:10.1007/978-3-319-03850-6_15`.

18 Luc Segoufin and Szymon Toruńczyk. Automata based verification over linearly ordered data domains. In Thomas Schwentick and Christoph Dürr, editors, *28th International Symposium on Theoretical Aspects of Computer Science, STACS 2011, March 10-12, 2011, Dortmund, Germany*, volume 9 of *LIPIcs*, pages 81–92. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011. URL: `http://dx.doi.org/10.4230/LIPIcs.STACS.2011.81`, `doi:10.4230/LIPIcs.STACS.2011.81`.

19 Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *Proceedings of the 14th International Parallel & Distributed Processing Symposium (IPDPS'00), Cancun, Mexico, May 1-5, 2000*, pages 263–268. IEEE Computer Society, 2000. URL: `http://dx.doi.org/10.1109/IPDPS.2000.845994`, `doi:10.1109/IPDPS.2000.845994`.

20 Nir Shavit and Asaph Zemach. Scalable concurrent priority queue algorithms. In Brian A. Coan and Jennifer L. Welch, editors, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing, PODC, '99Atlanta, Georgia, USA, May 3-6, 1999*, pages 113–122. ACM, 1999. URL: `http://doi.acm.org/10.1145/301308.301339`, `doi:10.1145/301308.301339`.

21 Viktor Vafeiadis. Automatically proving linearizability. In *CAV '10*, volume 6174 of *LNCS*, pages 450–464.

22 Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *POPL '86: Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–193. ACM Press, 1986.