Checking Linearizability of Concurrent Priority Queues

Ahmed Bouajjani¹, Constantin Enea¹, and Chao Wang¹

¹ Institut de Recherche en Informatique Fondamentale, {abou, cenea, wangch}@irif.fr

Abstract

Efficient implementations of concurrent objects such as atomic collections are essential to modern computing. Programming such objects is error prone: in minimizing the synchronization overhead between concurrent object invocations, one risks the conformance to sequential specifications – or in formal terms, one risks violating linearizability. Unfortunately, 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 such as task scheduling or discrete event simulation, and show that verifying linearizability of such implementations can be reduced to control-state reachability. This reduction entails the first decidability results for verifying concurrent priority queues in the context of an unbounded number of threads, and it enables the application of existing safety-verification tools for establishing their correctness.

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Modern computer software is increasingly concurrent. Interactive applications and services necessitate reactive asynchronous operations to handle requests immediately as they happen, rather than waiting for long-running operations to complete. Furthermore, as processor manufacturers approach clock-speed limits, performance improvements are more-often achieved by parallelizing operations across multiple processor cores.

Multithreaded software is typically built with specialized “concurrent objects” like atomic integers, queues, maps, priority queues. These objects’ methods are designed to conform to better established sequential specifications, a property known as linearizability [14], despite being optimized to avoid blocking and exploit parallelism, e.g., by using atomic 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 [1]. However, recent work [5] has shown that for particular classes of objects, i.e., 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] [7] [16] [20] [19], and concrete implementations exist in many modern languages like C++ or Java. Priority queues are collections providing \texttt{put} and \texttt{rm} methods for adding and removing values. Every value added 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 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 a queue, the 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 $\text{put}(a, p_1) \cdot \text{put}(b, p_3) \cdot \text{put}(c, p_2) \cdot \text{rm}(a, p_1) \cdot \text{rm}(c, p_2)$ 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.

Following the approach in [5], 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. However, differently from the case of stacks and queues where finite-state automata are sufficient, we show that the case of priority queues needs register automata where registers are used to store and compare priorities.

This characterization is obtained in several steps. We first define a recursive procedure which recognizes valid sequential executions which is then extended to recognize linearizable concurrent executions. Intuitively, for an input execution $e$, this procedure handles values occurring in $e$ one by one, starting with values of maximal priority (which are 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 same is done for 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 [15,8,18] (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 that are satisfied by implementations of this ADT, and formalizes the notion of linearizability. Sections 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.

### 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 (using a call to 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. In case multiple values are assigned with the same priority, rm returns the least recent value (according to a FIFO semantics). Also, when the set of values stored in the priority queue is empty, rm returns the distinguished value empty. Concurrent implementations of the priori-
ity queue allow the methods *put* and *rm* to be called concurrently from different threads. However, every method invocation should give the illusion that it takes place instantaneously at some point between its invocation and its return. We formalize (concurrent) executions and implementations in Section 2.1. Section 2.2 introduces a set of properties satisfied by all the implementations we are aware of, and Section 2.3 defines the standard correctness criterion for concurrent implementations of ADTs known as *linearizability* [14].

### 2.1 Executions

We fix a (possibly infinite) set $D$ of data values, a (possibly infinite) set $P$ of priorities, a partial order $\prec$ among elements in $P$, and an infinite set $O$ 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 $a \in D$, $a' \in D \cup \{\text{empty}\}$, $p \in P$, and $o \in O$, 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. 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 (i.e., 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 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. To ease the reading, 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 (where $o \in O$). 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 labelled transition system (LTS). An LTS is a tuple $A = (Q, \Sigma, \rightarrow, q_0)$, where $Q$ is a set of states, $\Sigma$ is an alphabet of transition labels, $\rightarrow \subseteq Q \times \Sigma \times Q$ is a transition relation and $q_0$ is the initial state. We model priority queue as an LTS $PQ$ defined as follows: each state of the LTS $PQ$ is a mapping associating priorities in $P$ with sequences of values in $D$ representing a snapshot of the priority queue (for each priority, the values are ordered as they were inserted); the transition labels are $put(a, p)$ and $rm(a)$ operations; Each transition modifies the state as expected. For example, $q_1 \xrightarrow{\text{rm(\text{empty})}} q_2$ if $q_1 = q_2$, and $q_1$ and $q_2$ map each priority to the empty sequence $\epsilon$. Then, SeqPQ is the set of traces of $PQ$. The detailed definition of $PQ$ can be found in Appendix A.

An implementation $I$ 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 implementation methods can always make progress in isolation: 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.

---

1 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.
2.2 Semantic Properties of Priority Queues

We define two properties which are satisfied by priority queue implementations and which are important for the results that follow: (1) data independence [23] states that priority queue behaviors do not depend on the actual values which are added to the queue, and (2) closure under projection [5] states that intuitively, remove operations can return the same values no matter how many other different values are in the queue, assuming they don’t have more important priorities.

An execution \( e \) is data-differentiated if every value is added at most once, i.e., for each \( d \in D \), \( e \) contains at most one action \( \text{call} \) \((\text{put}, d, p)\) with \( o \in \emptyset \) and \( p \in P \). 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 \( E_{d} \).

A renaming function \( r \) is a function from \( D \) to \( D \). 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 keep 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 \( e' \in E_{d} \) 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. \( \text{SeqPQ} \) is data independent.

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

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

The proof of the following lemma can be found in Appendix B.

Lemma 3. \( \text{SeqPQ} \) is closed under projection, i.e., \( \text{proj}(e) \subseteq \text{SeqPQ} \) for each \( e \in \text{SeqPQ} \).

2.3 Linearizability

We recall the notion of linearizability [14] which is the de facto standard correctness condition for concurrent data structures. Given an execution \( e \), the happens-before relation \( <_{\text{hb}} \) between operations \( \circ \) is defined as follows: \( o_{1} <_{\text{hb}} o_{2} \), 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 \( o_{1} <_{\text{hb}} o_{2} \) and \( o_{3} <_{\text{hb}} o_{4} \), then either \( o_{1} <_{\text{hb}} o_{4} \), or \( o_{3} <_{\text{hb}} o_{2} \). Intuitively, this comes from the fact that concurrent threads share a notion of global time.

\[\text{In general, we refer to operations using their identifiers.}\]
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) \( o \) and \( f(o) \) is the same operation and (2) if \( o_1 \prec_{hb} o_2 \), then \( f(o_1) \prec_{hb} f(o_2) \). 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 (see Appendix B). This is similar to that in \cite{15}, where they use the notion of data-independence in \cite{23}. Section 5 will focus on characterizing linearizability for data-differentiated executions.

\begin{lemma}
A data-independent implementation \( \mathcal{I} \) is linearizable w.r.t a data-independent set \( S \) of sequential executions, if and only if \( \mathcal{I}_E \) is linearizable w.r.t. \( S_E \).
\end{lemma}

\section{Checking Linearizability of Priority Queue Executions}

We define a recursive procedure for checking linearizability of an 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.

\subsection{Characterizing Data-Differentiated Sequential Executions}

Checking whether a data-differentiated sequential execution belongs to SeqPQ could be implemented by checking membership into the set of traces of the LTS \( PQ \). The recursive procedure Check-PQ-Seq outlined in Algorithm 1 is an alternative to this membership test. 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:

\[
\text{put}(c, p_2) \cdot \text{put}(a, p_1) \cdot \text{rm}(a) \cdot \text{rm}(c) \cdot \text{rm}(\text{empty}) \cdot \text{put}(d, p_2) \cdot \text{put}(f, p_3) \cdot \text{rm}(f) \cdot \text{put}(b, p_1)
\]  

(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 \( \text{rm}(\text{empty}) \) operations are read-only (they don’t affect the state of the queue), they are selected first. Ensuring that an operation \( o = \text{rm}(\text{empty}) \) is correct boils down to checking that every \( \text{put}(x, p) \) operation before \( o \) is matched to a \( \text{rm}(x) \) operation which also occurs before \( o \). This is true in this case for \( x \in \{ a, c \} \). Therefore, the correctness of (1) reduces to the correctness of

\[
\text{put}(c, p_2) \cdot \text{put}(a, p_1) \cdot \text{rm}(a) \cdot \text{rm}(c) \cdot \text{put}(d, p_2) \cdot \text{put}(f, p_3) \cdot \text{rm}(f) \cdot \text{put}(b, p_1)
\]

When there are no more \( \text{rm}(\text{empty}) \) operations, the procedure selects a \( \text{put} \) operation adding a value with maximal priority which is not removed, and then a pair of \( \text{put} \) and \( \text{rm} \) operations adding and removing the same maximal priority value. For instance, since \( p_2 \) is a maximal priority, it selects the operation \( \text{put}(d, p_2) \). This operation is correct if \( d \) is the last value with priority \( p_2 \), and the correctness of (1) reduces to the correctness of

\[
\text{put}(c, p_2) \cdot \text{put}(a, p_1) \cdot \text{rm}(a) \cdot \text{rm}(c) \cdot \text{put}(f, p_3) \cdot \text{rm}(f) \cdot \text{put}(b, p_1)
\]

\footnote{An \( \text{rm}(o) \)-operation in an execution \( e \) is an operation identifier \( o \) s.t. \( e \) contains the actions \( \text{call}_o(m, a) \), \( \text{return}_o(m, a) \).}
Algorithm 1: Check-PQ-Seq

Input: A data-differentiated sequential execution $e$
Output: true iff $e \in \text{SeqPQ}$

1. if $e = \epsilon$ then
   2. return true;
3. if Has-EmptyRemoves($e$) then
   4. if $\exists o = rm(\text{empty}) \in e$ such that EmptyRemove-Seq($e, o$) holds then
      5. return Check-PQ-Seq($e \setminus o$);
   6. else if Has-UnmatchedMaxPriority($e$) then
      7. if $\exists x \in \text{values}($ holds then
         8. return Check-PQ-Seq($e \setminus x$);
      9. else if $\exists x \in \text{values}($ holds then
         10. return Check-PQ-Seq($e \setminus x$);
         11. return false;

Since there is no other value of maximal priority which is not removed, the procedure selects a pair of put, $rm$ operations with an argument of maximal priority $p_2$, for instance, put($c, p_2$) and $rm(c)$. The value returned by $rm$ 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.

In formal terms, the operations which are selected depend on the following set of predicates on executions:

- $\text{Has-EmptyRemoves}(e) = \text{true iff } e \text{ contains a } rm(\text{empty}) \text{ operation}$
- $\text{Has-UnmatchedMaxPriority}(e) = \text{true iff } p \in \text{unmatched-priorities}(e) \text{ for a maximal priority}$
  
  where $\text{priorities}(e)$, resp., $\text{unmatched-priorities}(e)$, 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 $\text{Has-MatchedMaxPriority}(e) = \neg \text{Has-EmptyRemoves}(e) \land \neg \text{Has-UnmatchedMaxPriority}(e)$. By an abuse of notation, we also assume that $\text{Has-UnmatchedMaxPriority}(e) \Rightarrow \neg \text{Has-EmptyRemoves}(e)$ (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:

- $\text{EmptyRemove-Seq}(e, o) = \text{true iff } e = u \cdot o \cdot v \text{ and matched}(u)$
- $\text{UnmatchedMaxPriority-Seq}(e, x) = \text{true iff } e = u \cdot \text{put}(x, p) \cdot v, p \not\in \text{priorities}(u \cdot v)$, and $p \not\in \text{priorities}(v)$
- $\text{MatchedMaxPriority-Seq}(e, x) = \text{true iff } e = u \cdot \text{put}(x, p) \cdot v \cdot \text{rm}(x) \cdot w, p \not\in \text{priorities}(u \cdot v)$,
  
  where $p < \text{priorities}(e)$ when $p < p'$ for some $p' \in \text{priorities}(e)$ (and similarly for $p < \text{unmatched-priorities}(e)$) or $p \not\leq \text{unmatched-priorities}(e)$, matched$_{\omega}(e, p)$ holds when each value with priority strictly smaller than $p$ is removed in $e$, and matched$_{\omega}(e, p)$ holds when matched$_{\omega}(e, p)$ holds for each
4. In general, there may exist multiple ways of choosing linearization points to define the same linearization. Our construction is agnostic to this choice.
- 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 \( l_1^v \), \( l_2^u \) and \( l_3^v \) 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 \( \text{put}(z_1, p_2) \cdot \text{rm}(z_2) \) which is incorrect.

- We define the sets of operations \( U', V' \) and \( W' \) such that \( l_1^v, l_2^u \) and \( l_3^v \) are the projections of \( l' \) to \( U', V' \), and \( W' \), respectively. This is done in two steps:

- 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 \( \text{ret}(\text{rm}, x) \). We add to \( W' \) the operation \( o \) and all the p-incomparable operations occurring after \( o \) in \( l' \). In this example, \( o \) is \( \text{rm}(z_1) \) and the only p-incomparable operation occurring after \( o \) in \( l' \) is \( \text{rm}(z_2) \) (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 \( \text{ret}(\text{put}, x, p) \), and (2) other \( \text{put} \) operations with priority \( p \). \( V' \) contains the remaining operations. In this example, \( U' \) contains \( \text{put}(z_1, p_2) \) and \( \text{put}(x_2, p) \).

- In conclusion, we have that \( l_1^v = \text{put}(z_1, p_2) \cdot \text{put}(x_2, p), \), \( l_2^u = \text{put}(z_2, p_2) \cdot \text{rm}(x_2) \cdot \text{put}(y_1, p_1) \cdot \text{rm}(y_1), \) and \( l_3^v = \text{rm}(z_1) \cdot \text{rm}(z_2) \). Figure 1(b) draws linearization points for each operation in \( e \) defining the linearization \( l' \).

Section 3 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 \( e, e \subseteq \text{SeqPQ} \) if and only if for each \( e' \in \text{proj}(e) \), 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 w.r.t. $\Gamma = \forall \Gamma = \forall$ for ing all these operations results in a sequential execution which still satisfies this property. Similarly, ($\exists \Gamma$ execution. Then, $\Gamma$-complete automata for each $\Gamma$-linearizable, with self-loops that allow any operation with parameter $\top$ results in an automaton satisfying the following property called $\Gamma$-completeness.

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

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

**Definition 8.** For $\Gamma \in \{\text{EmptyRemove, UnmatchedMaxPriority, MatchedMaxPriority}\}$, an automaton $A$ is called $\Gamma$-complete when for each data-independent implementation $I$:

$A \cap I \neq \emptyset$ if and only if there exists $e \in I$ and $e' \in \text{proj}(e)$ such that $e'$ is not $\Gamma$-linearizable.

Section 4.1 describes a MatchedMaxPriority-complete automaton, the other automata being defined in Appendix D.1 and Appendix D.7 Therefore, the following holds.

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

When defining $\Gamma$-complete automata, we assume that every implementation $I$ 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 $e \in I$ and every priority $p \in P$, the projection of $e$ to values with priority $p$ is linearizable (w.r.t. SeqPQ). This property can be checked separately using register automata obtained from the finite automata in [5] for FIFO queue (see Appendix D.1 for more details). Note that this assumption excludes some obvious violations, such as a $\text{rm}(a)$ operation happens before a $\text{put}(a, p)$ operation, for some $p$.

Also, we consider $\Gamma$-complete automata for $\Gamma \in \{\text{UnmatchedMaxPriority, MatchedMaxPriority}\}$, 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 $e$, $e|_{\leq p}$ is the projection of $e$ to the set of values with priorities smaller than $p$. Then,

**Lemma 10.** Let $\Gamma \in \{\text{UnmatchedMaxPriority, MatchedMaxPriority}\}$ and $e$ a data-differentiated execution. Then, $e$ is $\Gamma$-linearizable if $e|_{\leq p}$ is $\Gamma$-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. $l = u \cdot \text{put}(x, p) \cdot v \cdot \text{rm}(x) \cdot w \in \text{MatchedMaxPriority-Seq}(s, x)$. Since MatchedMaxPriority-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 $\Gamma = \text{UnmatchedMaxPriority}$.

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
Checking Linearizability of Concurrent Priority Queues

has a linearization \( l_2 \), then the execution has a linearization which is defined as an interleaving of \( l_1 \) and \( l_2 \) (see Appendix D.2 for more details).

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

The following shows that \( \Gamma \)-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.

\[ \textbf{Theorem 11.} \] Let \( \mathcal{I} \) be a data-independent implementation, and \( \Lambda(\Gamma) \) be a \( \Gamma \)-complete automaton for each \( \Gamma \). Then, \( \mathcal{I} \sqsubseteq \text{SeqPQ} \) if and only if \( \mathcal{I} \cap \Lambda(\Gamma) = \emptyset \) for all \( \Gamma \).

4.1 A MatchedMaxPriority-complete automaton

A differentiated execution \( e \) is not MatchedMaxPriority-linearizable when all the \text{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 \( \text{MatchedMaxPriority-Seq}(e, x) \) 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 \( \text{MatchedMaxPriority}^\ast \) the strengthening of \( \text{MatchedMaxPriority} \) with the condition that all the values other than \( x \) have a priority strictly smaller than \( p \) (corresponding to the first case), and by \( \text{MatchedMaxPriority}^- \) the strengthening of the same formula with the negation of this condition (corresponding to the second case). We use particular instances of register automata [15] [8] [13] 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 \( P \). The transition labels can be one of the following:

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

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

4.1.1 A MatchedMaxPriority\(^\ast\)-complete automaton

We give a typical example of an execution \( e \) which is not MatchedMaxPriority\(^-\)-linearizable in Figure 2. Intuitively, this is a violation because during the whole execution of \( \text{rm}(b) \), 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 \( \text{ret}(\text{put}, x, p) \) to the call of the matching remove \( \text{call}(\text{rm}, x) \), 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 \( \text{MatchedMaxPriority-Seq}(s, b) \) 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.

\[
\begin{align*}
\text{put}(b, p_1) & \quad \text{rm}(b) \\
\text{put}(d, p_1) & \quad \text{rm}(d) \\
\text{put}(e, p_2) & \quad \text{rm}(e) \\
\text{put}(f, p_2) & \quad \text{rm}(f)
\end{align*}
\]

\[\textbf{Figure 2}\] An execution that is not MatchedMaxPriority\textsuperscript{\textgreater\textless}-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.

\begin{definition}
\textbf{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:
\begin{itemize}
\item the nodes are the values occurring in $e$,
\item there is an edge from $d_1$ to $x$, if put$(d_1, _) <_{hb} \text{put}(x, p)$ or put$(d_1, _) <_{hb} \text{rm}(x)$,
\item there is an edge from $x$ to $d_1$, if rm$(x) <_{hb} \text{rm}(d_1)$ or rm$(d_1)$ does not exists,
\item there is an edge from $d_1$ to $d_2$, if put$(d_1, _) <_{hb} \text{rm}(d_2, _)$.\end{itemize}
\end{definition}

The execution in Figure 2 is not MatchedMaxPriority\textsuperscript{\textgreater\textless}-linearizable because the left-right constraint of the maximal priority value $b$ contains a cycle: $f \rightarrow c \rightarrow d \rightarrow b \rightarrow f$. The presence of such a cycle is equivalent to the execution not being MatchedMaxPriority\textsuperscript{\textgreater\textless}-linearizable (see Appendix D.3), as indicated by the following lemma:

\begin{lemma}
\textbf{Lemma 13.} Given a data-differentiated execution $e$ where Has-MatchedMaxPriority$(e)$ holds, let $p$ be its maximal priority and put$(x, p), \text{rm}(x)$ 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 $x$.
\end{lemma}

When the left-right constraint of the maximal priority value $x$ contains a cycle of the form $d_1 \rightarrow \ldots \rightarrow d_m \rightarrow x \rightarrow d_1$ for some $d_1, \ldots, d_n \in \mathbb{D}$, we say that $x$ is covered by $d_1, \ldots, d_n$. The shape of such a cycle (i.e., the alternation between call/return actions of put/\text{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\textsuperscript{\textgreater\textless}-complete automaton we need to consider several cases depending on the order between the call/return actions of the put/\text{rm} operations that add and respectively, remove the value $b$. For example, 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$(\text{put}, a, _)$. The actions with value $b$, which are pictured in Figure 4 (a) (this action cannot occur after call$(rm, b, _)$. We show in Figure 4 all possible orders of the call/return actions of adding and removing $b$, and also possible orders of the first ret$(\text{put}, a, _)$. In Appendix D.3 three register automata is constructed according to the cases of Figure 4 (b), (c) and (d), respectively.
Figure 3: Register automaton $A_{\text{lar}}^1$. We use the following notations: $C_1 = C \cup \{\text{ret}(\text{rm}, a)\}$, $C_2 = C \cup \{\text{call}(\text{put}, a, = r)\}$, $C_3 = C_2 \cup \{\text{ret}(\text{rm}, a)\}$, where $C = \{\text{call}(\text{put}, T, \text{true}), \text{ret}(\text{put}, T, \text{true}), \text{call}(\text{rm}, d), \text{ret}(\text{rm}, d), \text{call}(\text{rm}, \text{empty}), \text{ret}(\text{rm}, \text{empty})\}$.

Figure 4: Four cases of ordering actions with value $b$, and possible orders of the first $\text{ret}(\text{put}, a, = r)$ 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 $\text{rm}(b)$ are disabled by the position of $\text{rm}(a)$ 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 $\text{put}(a, p_4) <_{hb} \text{put}(b, p_4)$ and values of the same priority are removed in FIFO order, $\text{rm}(a)$ should be linearized before $\text{rm}(b)$ (i.e., this execution should be linearizable w.r.t. a sequence where $\text{rm}(a)$ occurs before $\text{rm}(b)$). Since $\text{rm}(b)$ 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 $\text{call}(\text{rm}, a)$, and thus $\text{rm}(a)$ cannot be linearized before $\text{rm}(b)$.

Figure 5: An execution that is not linearizable w.r.t. MatchedMaxPriority".

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 \prec_{pb} b if one of the following holds:
(1) \text{put}(a, \_)_h \prec_{hb} \text{put}(b, \_), (2) \text{rm}(a) \prec_{hb} \text{rm}(b), or (3) \text{rm}(a) \prec_{hb} \text{put}(b, \_). Sometimes we use \( \prec_{pb} \) to explicitly distinguish between these three cases. Let \( \prec_{pb} \) be the transitive closure of \( \prec_{pb} \).

Then, to model the time intervals in which a remove operation like \text{rm}(b) in Figure 5, can be linearized (outside of intervals of smaller priority values) we introduce the notion of gap-point. Here we assume that the index of actions of an execution starts from 0.

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

The case of Figure 5 can be formally described as follows: \( a \prec_{pb} b \) while the right-most gap-point of b is before \text{call}(\text{rm}, a) or \text{call}(\text{put}, a, p_4). \) The following lemma states that these conditions are enough to characterize non-linearizability w.r.t MatchedMaxPriority\(^m\) (see Appendix D.4).

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

To characterize violations to linearizability w.r.t. MatchedMaxPriority\(^m\) using an automaton that tracks a bounded number of values, we show that the number of values needed to witness that \( y \prec_{pb} x \) for some x and y is bounded.

\begin{lemma}
Let e be a data-differentiated execution such that \( a \prec_{pb} a_1 \prec_{pb} \ldots \prec_{pb} a_m \prec_{pb} b \) holds for some set of values \( a, a_1, \ldots, a_m, b \). Then, one of the following holds:
- \( a \prec_{pb} b, a \prec_{pb} b, \) or \( a \prec_{pb} b, \)
- \( a \prec_{pb} a_i \prec_{pb} b \) or \( a \prec_{pb} a_i \prec_{pb} b, \) for some i.
\end{lemma}

To characterize violations to linearizability w.r.t. MatchedMaxPriority\(^m\), one has to consider all the possible orders between \text{call}/\text{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 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, 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 \text{call}(\text{rm}, a) to \text{ret}(\text{rm}, b) is covered by lower priority values (and thus, there is no gap-point of b which occurs after \text{call}(\text{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 \( \top \).

\subsection{Decidability Result}

We describe a class C of data-independent implementations for which linearizability w.r.t. SeqPQ is decidable. The implementations in 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 \( \Gamma \)-complete automata \( A(\Gamma) \) uses a fixed set \( D = \)
Checking Linearizability of Concurrent Priority Queues

**Figure 6** Five cases that need to be considered.

**Figure 7** A case in deriving a MatchedMaxPriority\(^{-}\)-complete automaton. We use the following notations: \( C = \{ \text{call}(\text{put}, T, \text{true}), \text{ret}(\text{put}, T, \text{true}), \text{call}(\text{rm}, T), \text{ret}(\text{rm}, T), \text{call}(\text{rm, empty}), \text{ret}(\text{rm, empty}) \} \), \( C_1 = C \cup \{ \text{call}(\text{put}, d, < r) \} \), \( C_2 = C_1 \cup \{ \text{ret}(\text{put, b, = r}) \} \), \( C_3 = C_2 \cup \{ \text{ret}(\text{rm, d}) \} \), \( C_4 = C \cup \{ \text{ret}(\text{put, b, = r}), \text{ret}(\text{rm, d}) \} \).
\{a, b, a_1, d, e, ⊤\} of values, we have that \(C \cap A(Γ) \neq ∅\) for some \(Γ\) iff \(C_D \cap A(Γ) \neq ∅\) where \(C_D\) is the subset of \(C\) that uses only values in \(D\).

The set of executions \(C_D\) 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 \(C_D\), 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 [10].

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. The detailed proof can be found in Appendix D.8.

▶ Theorem 17. Verifying whether an implementation in \(C\) 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 [11] 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, 12], 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] [13, 22]. The approach in [22] 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 [13] 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 [9] 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

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

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

7. 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

8. 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/bfb0055056

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

10. 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.

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

12. Jad Hamza. On the complexity of linearizability. In NETYS ’15. Springer, 2015.

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

14. Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst., 12(3):463–492, 1990.
Ahmed Bouajjani, Constantin Enea, and Chao Wang

15 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

16 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

17 Yujie Liu and Michael F. Spear. A lock-free, array-based priority queue. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012, pages 323–324, 2012.

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

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 Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings, page 84. IEEE Computer Society, 2003.

22 Viktor Vafeiadis. Automatically proving linearizability. In CAV ’10, volume 6174 of LNCS, pages 450–464.

23 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.
A labelled transition system (LTS) is a tuple $\mathcal{A} = (Q, \Sigma, \rightarrow, q_0)$, where $Q$ is a set of states, $\Sigma$ is an alphabet of transition labels, $\rightarrow \subseteq Q \times \Sigma \times Q$ is a transition relation and $q_0$ is the initial state.

Let us model priority queue as an LTS $PQ = (Q, \rightarrow, q_0)$ as follows:

- Each state of $Q$ is a function from $P$ into a finite sequence over $\mathbb{D}$.
- The initial state $q_0$ is a function that maps each element in $P$ into $\epsilon$.
- $\Sigma = \{ \text{put}(a, p), \text{rm}(a), \text{rm}(\text{empty}) | a \in \mathbb{D}, p \in P \}$.
- The transition relation $\rightarrow$ is defined as follows:

  - $q_1 \overset{\text{put}(a, p)}{\rightarrow} q_2$, if $q_1$ maps $p$ into some finite sequence $l$, and $q_2$ is the same as $q_1$, except for $p$, where it maps $p$ into $a \cdot l$.
  - $q_1 \overset{\text{rm}(a)}{\rightarrow} q_2$, if $q_1$ maps $p$ into $l \cdot a$ for some finite sequence $l$, and $q_2$ is the same as $q_1$, except for $p$, where it maps $p$ into $l$. We also require that for each priority $p'$ such that $p' \prec p$, $q_1$ and $q_2$ map $p'$ into $\epsilon$.
  - $q_1 \overset{\text{rm}(\text{empty})}{\rightarrow} q_2$, if $q_1 = q_2$, and they maps each element in $P$ into $\epsilon$.

A path of an LTS is a finite transition sequence $q_0 \overset{\beta_1}{\rightarrow} q_1 \overset{\beta_2}{\rightarrow} \ldots \overset{\beta_k}{\rightarrow} q_k$ for $k \geq 0$, where $q_0$ is the initial state of the LTS. A trace of an LTS is a finite sequence $\beta_1 \cdot \beta_2 \cdot \ldots \cdot \beta_k$, where $k \geq 0$ if there exists a path $q_0 \overset{\beta_1}{\rightarrow} q_1 \overset{\beta_2}{\rightarrow} \ldots \overset{\beta_k}{\rightarrow} q_k$ of the LTS. Let SeqPQ be the set of traces of $PQ$. The following lemma states that SeqPQ is indeed the set of sequences obtained by renaming sequences accepted by Check-PQ-Seq. Given $\Gamma \in \{ \text{EmptyRemove}, \text{UnmatchedMaxPriority}, \text{MatchedMaxPriority} \}$, let us use $l_2 \xrightarrow{\Gamma} l_1$ to mean that when we use Check-PQ-Seq to check $l_2$, we choose the branch of Has-$\Gamma$ and finally recursively call Check-PQ-Seq to check $l_1$.

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

**Proof.** Let SeqPQ$_f$ be the set of data-differentiated sequences, such that $\epsilon \in \text{SeqPQ}_f$, if Check-PQ-Seq($\epsilon$) = true. We need to prove that SeqPQ$_f = \text{SeqPQ}_\#$. We prove SeqPQ$_f \subseteq \text{SeqPQ}_\#$ by induction.

- It is obvious that $\epsilon \in \text{SeqPQ}_\#$.
- If $l_1 \in \text{SeqPQ}_\#$ and $l_2 \overset{\text{MatchedMaxPriority}}{\rightarrow} l_1$, then we need to prove that $l_2 \in \text{SeqPQ}_\#$. We know that $l_2 = u \cdot v \cdot w$, such that MatchedMaxPriority-Seq($l_2, x$) holds and $l_2 = u \cdot \text{put}(x, p) \cdot v \cdot \text{rm}(x) \cdot w$. Assume that $u = \alpha_1 \cdot \ldots \cdot \alpha_i$, $v = \alpha_{i+1} \cdot \ldots \cdot \alpha_j$ and $w = \alpha_{j+1} \cdot \ldots \cdot \alpha_m$. Assume that $q_0 \overset{\alpha_1}{\rightarrow} q_1 \ldots \overset{\alpha_i}{\rightarrow} q_i \overset{\alpha_{i+1}}{\rightarrow} q_{i+1} \ldots \overset{\alpha_j}{\rightarrow} q_j \overset{\alpha_{j+1}}{\rightarrow} q_{j+1} \ldots \overset{\alpha_m}{\rightarrow} q_m$ is the path of $l_1$ on PQ. For each $i \leq k \leq j$, let $q_k'$ be the same as $q_k$, except that $q_k'$ maps $p$ into $x \cdot l_k$ and $q_k$ maps $p$ into $l_k$ for some finite sequence $l_k$.

We already know that $q_0 \overset{\alpha_1}{\rightarrow} q_1 \ldots \overset{\alpha_i}{\rightarrow} q_i$, and it is obvious that $q_i \overset{\text{put}(x, p)}{\rightarrow} q_i'$. Since (1) all put with priority $p$ is in $u$, and (2) in $u \cdot v$, only values with priority either incomparable, or less, or equal than $p$ is removed, we can see that it is safe to add to each $q_k$ ($1 \leq k \leq j$) with a newest $x$ with priority $p$. Or we can say, $q_i' \overset{\alpha_{i+1}}{\rightarrow} q_{i+1}' \ldots \overset{\alpha_j}{\rightarrow} q_j'$ are transitions of $PQ$. Since matched$_\#(u \cdot v \cdot p)$ holds, we can see that $q_i$ maps each priority that is smaller than $p$ into $\epsilon$ and maps $p$ into $\epsilon$, and $q_i'$ maps each priority that is smaller than $p$ into $\epsilon$ and maps $p$ into $\epsilon$. Then,
we can see that \( q_j^l \xrightarrow{rm(x)} q_j \). We already know that that \( q_j \xrightarrow{\alpha_{i+1}} q_{j+1} \ldots \xrightarrow{\alpha_m} q_m \). Therefore, we can see that \( l_2 = u \cdot \text{put}(x, p) \cdot v \cdot rm(x) \in \text{SeqPQ} \).

- If \( l_1 \in \text{SeqPQ}_d \) and \( l_2 \xrightarrow{\text{UnmatchedMaxPriority}} l_1 \). Then we need to prove that \( l_2 \in \text{SeqPQ} \). We know that \( l_1 = u \cdot v \cdot \), such that UnmatchedMaxPriority-Seq\((l_2, x)\) holds and \( l_2 = u \cdot \text{put}(x, p) \cdot \). Assume that \( u = \alpha_1 \ldots \alpha_i \) and \( v = \alpha_{i+1} \ldots \alpha_m \). Assume that \( q_0 \xrightarrow{\alpha_i} q_1 \ldots \xrightarrow{\alpha_{i+1}} q_{i+1} \ldots \xrightarrow{\alpha_m} q_m \) is the path of \( l_1 \) on PQ. For each \( i \leq k \leq m \), let \( q_k' \) be the same as \( q_k \), except that \( q_k' \) maps \( p \) into \( x \cdot l_k \) and \( q_k' \) maps \( p \) into \( l_k \) for some finite sequence \( l_k \).

  We already know that \( q_0 \xrightarrow{\alpha_1} q_1 \ldots \xrightarrow{\alpha_i} q_i \), and it is obvious that \( q_i \xrightarrow{\text{put}(x, p)} q_i'. \) Since (1) all put with priority \( p \) is in \( u \), (2) in \( u \cdot v \), only values with priority either incomparable, or less, or equal than \( p \) is removed, we can see that it is safe to add to each \( q_k \) \( (1 \leq k \leq m) \) with a newest \( x \) with priority \( p \). Or we can say, \( q_0, q_1, q_2, \ldots, q_m \) are transitions of PQ. Therefore, we can see that \( l_2 = u \cdot \text{put}(x, p) \cdot v \in \text{SeqPQ} \).

- If \( l_1 \in \text{SeqPQ}_d \) and \( l_2 \xrightarrow{\text{EmptyRemove}} l_1 \). Then we need to prove that \( l_2 \in \text{SeqPQ} \). We know that \( l_1 = u \cdot v \cdot \), such that EmptyRemove-Seq\((l_2, o)\) holds, and \( l_2 = u \cdot rm(\text{empty}) \cdot v \) with \( o = rm(\text{empty}) \).

  Assume that \( u = \alpha_1 \ldots \alpha_i \) and \( v = \alpha_{i+1} \ldots \alpha_m \). Assume that \( q_0 \xrightarrow{\alpha_i} q_1 \ldots \xrightarrow{\alpha_i} q_i \ldots \xrightarrow{\alpha_{i+1}} q_{i+1} \ldots \xrightarrow{\alpha_m} q_m \) is the path of \( l_0 \) on PQ. We already know that \( q_0 \xrightarrow{\alpha_i} q_1 \ldots \xrightarrow{\alpha_i} q_i \). Since matched\((u)\) holds, we can see that \( q_i \) maps each element in \( p \) into \( e \), and then \( q_i \xrightarrow{\text{empty}} q_i \). We already know that \( q_0 \xrightarrow{\alpha_i} q_1 \ldots \xrightarrow{\alpha_i} q_i \ldots \xrightarrow{\alpha_{i+1}} q_{i+1} \ldots \xrightarrow{\alpha_m} q_m \). Therefore, we can see that \( l_2 = u \cdot \text{rm}(\text{empty}) \cdot v \in \text{SeqPQ} \).

To prove that \( \text{SeqPQ}_d \subseteq \text{SeqPQ}_f \), we show that given \( l_2 \in \text{SeqPQ}_d \), how to construct a sequence \( l_1 \), such that \( l_2 \xrightarrow{\Gamma} l_1 \) for some \( \Gamma \), and \( l_1 \in \text{SeqPQ} \). Based on this, we can decompose a sequence of SeqPQ into \( e \), and this process ensures that this sequence is in \( \text{SeqPQ}_f \). Note that from a \( l_2 \) we may construct more than one \( l_1 \), and this does not influence the correctness of our proof.

- If Has-EmptyRemoves\((l_2)\): Assume that \( l_2 = u \cdot \text{rm}(\text{empty}) \cdot v \). It is easy to see that matched\((u)\) holds. Let \( l_1 = u \cdot v \). It is easy to see that \( l_2 \xrightarrow{\text{EmptyRemove}} l_1 \), and UnmatchedMaxPriority-Seq\((l_2, o)\) holds for some \( o = rm(\text{empty}) \).

  Assume that \( u = \alpha_1 \ldots \alpha_i \) and \( v = \alpha_{i+1} \ldots \alpha_m \). Since We already know that \( q_0 \xrightarrow{\alpha_i} q_1 \ldots \xrightarrow{\alpha_i} q_i \ldots \xrightarrow{\alpha_{i+1}} q_{i+1} \ldots \xrightarrow{\alpha_m} q_m \) is transitions of PQ. It is easy to see that \( q_0 = q'_0 \), and they map each element in \( p \) into \( e \). Then we can see that \( q_0 \xrightarrow{\alpha_i} q_1 \ldots \xrightarrow{\alpha_i} q_i \ldots \xrightarrow{\alpha_{i+1}} q_{i+1} \ldots \xrightarrow{\alpha_m} q_m \) is transitions of \( PQ \), and \( l_1 \in \text{SeqPQ} \).

- If Has-UnmatchedMaxPriority\((l_2)\): Assume that \( l_2 = u \cdot \text{put}(x, p) \cdot v \), such that all put with priority \( p \) of \( u \cdot v \) is in \( u \). Let \( l_1 = u \cdot v \). According to construction of PQ, we can see that UnmatchedMaxPriority-Seq\((l_2, x)\) holds, and \( l_2 \xrightarrow{\text{UnmatchedMaxPriority}} l_1 \).

  Assume that \( u = \alpha_1 \ldots \alpha_i \) and \( v = \alpha_{i+1} \ldots \alpha_m \). We already know that \( \text{pa} = q_0 \xrightarrow{\alpha_i} q_1 \ldots \xrightarrow{\alpha_i} q_i \ldots \xrightarrow{\alpha_{i+1}} q_{i+1} \ldots \xrightarrow{\alpha_m} q_m \) are transitions of PQ. For each \( i+1 \leq k \leq m \), let \( q_k' \) be the same as \( q_k \), except that \( q_k' \) maps \( p \) into some \( x \cdot l_k \) for some finite sequence \( l_k \), and \( q_k' \) maps \( p \) into \( l_k \). Since (1) all put with priority \( p \) of \( u \cdot v \) is in \( u \) and (2) \( p \) is one of maximal priority of \( l_2 \), it is safe to remove \( x \) without influence other transitions of \( pa \). Or we can say, \( q_0 \xrightarrow{\alpha_i} q_1 \ldots \xrightarrow{\alpha_i} q_i \ldots \xrightarrow{\alpha_{i+1}} q_{i+1} \ldots \xrightarrow{\alpha_m} q_m \) are transitions of PQ. Therefore, \( l_1 \in \text{SeqPQ} \).

- If Has-MatchedMaxPriority\((l_2)\): Assume that \( l_2 = u \cdot \text{put}(x, p) \cdot v \cdot \text{rm}(x) \cdot w \), such that all put with priority \( p \) of \( u \cdot v \cdot w \) is in \( u \). Let \( l_1 = u \cdot v \cdot w \). According to construction of PQ, we can see that MatchedMaxPriority-Seq\((l_2, x)\) holds, and \( l_2 \xrightarrow{\text{MatchedMaxPriority}} l_1 \).

  Assume that \( u = \alpha_1 \ldots \alpha_i \) and \( v = \alpha_{i+1} \ldots \alpha_j \) and \( w = \alpha_{j+1} \ldots \alpha_m \). We already know that \( q_0 \xrightarrow{\alpha_i} q_1 \ldots \xrightarrow{\alpha_i} q_i \ldots \xrightarrow{\alpha_{i+1}} q_{i+1} \ldots \xrightarrow{\alpha_j} q_j \ldots \xrightarrow{\alpha_{m}} q_m \).
For each \( i+1 \leq k \leq j \), let \( q_k' \) be the same as \( q_k \), except that \( q_k \) maps \( p \) into \( x \cdot l_k \) for some finite sequence \( l_k \), and \( q_k' \) maps \( p \) into \( l_k \). Since (1) \( p \) is one of maximal priority in \( l_2 \), (2) \( x \) is the newest value with priority \( p \) in \( l_2 \), and (3) \( x \) is not removed until \( rm(x) \), we know that whether we keep \( x \) or remove it will not influence transitions from \( q_{i+1} \) to \( q_j \). Then we can see that \( q_0 \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_i} q_i \xrightarrow{\alpha_{i+1}} q_{i+1} \xrightarrow{\alpha_{i+2}} q_{j} \xrightarrow{\alpha_{j+1}} \ldots \xrightarrow{\alpha_m} q_m \) are transitions of \( PQ \). Therefore, \( l_1 \in \text{SeqPQ} \).

This completes the proof of this lemma.

\[ \blacktriangleright \quad \blacktriangleright \]

**B Proofs in Section 2**

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

**Proof.** By Lemma 5, SeqPQ is equivalent to the set of sequences obtained by renaming sequences accepted by \( \text{Check-PQ-Seq} \). It is easy to see that for the predicates of \( \text{Check-PQ-Seq} \), if a sequential execution satisfy it, then its sub-sequence also satisfy it. For example, if \( \text{matched}(u) \) holds, then \( \text{matched}(u|D) \) holds for each set \( D \) of values. This completes the proof of this lemma.

\[ \blacktriangleright \quad \blacktriangleright \]

**Lemma 4** A data-independent implementation \( I \) is linearizable w.r.t a data-independent set \( S \) of sequential executions, if and only if \( I_\neq \) is linearizable w.r.t. \( S_\neq \).

**Proof.** To prove the only if direction, given a data-differentiated execution \( e \in I_\neq \). By assumption, it is linearizable with respect to a sequential execution \( l \in S \), and the bijection between the operations of \( e \) and the operations of \( l \) ensures that \( l \) is differentiated and belongs to \( S_\neq \).

To prove the if direction, given an execution \( e \in I \). By data independence of \( I \), we know that there exists \( e' \in I_\neq \) and a renaming function \( r \), such that \( r(e') = e \). By assumption, \( e' \) is linearizable with respect to a sequential execution \( l' \in S_\neq \). Let \( l = r(l') \). By data independence of \( S \) it is easy to see that \( l \in S \), and it is easy to see that \( e \sqsubseteq l \) using the same bijection used for \( e' \sqsubseteq l' \).

\[ \blacktriangleright \quad \blacktriangleright \]

**C Proofs in Section 3**

### C.1 Definition of Step-by-Step linearizability

We introduce the notion of step-by-step linearizability, which means that from a linearization of \( e \setminus x \) that satisfy the requirements of priority queue, we can obtain a linearization of \( e \) that satisfy the requirements of priority queue. Its formal definition is as follows:

**Definition 18.** Given \( \Gamma \in \{ \text{EmptyRemove}, \text{UnmatchedMaxPriority}, \text{MatchedMaxPriority} \} \), \( \Gamma \) is step-by-step linearizability, if for each data-differentiated execution \( e \) where \( \Gamma \cdot \text{conc}(e, \alpha) \) for some \( \alpha \), then \( e \setminus \alpha \sqsubseteq \text{SeqPQ} \Rightarrow e \sqsubseteq \text{SeqPQ} \).

\( \text{SeqPQ} \) is step-by-step linearizability, if each \( \Gamma \) is step-by-step linearizability.

Our notion of step-by-step linearizability is inspired by the step-by-step linearizability in [5].

Given a data-differentiated execution \( e \), we can obtain a sequence \( e' \) from \( e \) by adding \( \text{put}(a, p) \) (resp., \( \text{rm}(a) \), \( \text{rm}(\text{empty}) \)) between each pair of \( \text{call}(\text{put}, a, p) \) and \( \text{ret}(\text{put}, a, p) \) (resp., \( \text{call}(\text{rm}, a) \) and \( \text{ret}(\text{rm}, a) \), \( \text{call}(\text{rm}, \text{empty}) \) and \( \text{ret}(\text{rm}, \text{empty}) \)). Such \( e' \) is called an execution with linearization points, and we call the projection of \( e' \) into \( m(a, b) \) the linearization of \( e \).
C.2 Obtaining New Sequences while Ensuring that they are in SeqPQ

Before we prove the step-by-step linearizability of SeqPQ, we introduce several lemmas, which are used to ensure some sub-sequences of SeqPQ still belong to SeqPQ. Given a data-differentiated sequence \( l \) and one of its maximal priority \( p \), let \( O_c(l, p) \) and \( O_i(l, p) \) be the set of operations with priorities comparable with \( p \) and incomparable with \( p \) in \( l \), respectively. Similarly we can define \( D_c(l, p) \) and \( D_i(l, p) \) for set of values instead of set of operations. We can see that each priority of values in \( O_i(l, p) \) is either larger or incomparable with priorities of values in \( O_c(l, p) \).

The following lemma shows that if a new sequence is generated by erasing some operations in \( O_c(l, p) \) while keeping the remaining \( O_c(l, p) \) sub-sequences in SeqPQ, then this new sequence is still in SeqPQ. Note that this is different from projection on value.

**Lemma 19.** Given a data-differentiated sequential execution \( l \in SeqPQ \) and a maximal priority \( p \) in \( l \), where \( l \) does not contain \( rm(\text{empty}) \). Let \( l' \) be generated from \( l \) by discarding some operations in \( O_c(l, p) \), and \( l'_p \subset O_i(l, p) \in SeqPQ \). Then, \( l' \in SeqPQ \).

**Proof.** Let \( l = o_1, \ldots, o_m \), and \( q_0 \xrightarrow{a} q_1 \xrightarrow{a} \ldots \xrightarrow{a} q_m \) be the path of \( l \) in \( PQ \). Assume that \( l' \) is generated from \( l \) by discarding \( o_{indj}, \ldots, o_{indn} \). Let \( D \) be the set such that \( D \) contains \( a \), if \( put(a, \_ ) \) in \( o_{indj}, \ldots, o_{indn} \). For each \( i \), let \( q'_j \) be generated from \( q_j \) by erasing values in \( D \).

For each \( q'_j \) with \( j \neq \text{ind}_1 \), if \( o_{j+1} \) is \( \text{put} \), then it is obvious that \( q'_j \xrightarrow{a} q'_{j+1} \). Else, assume \( o_{j+1} = \text{rm}(a) \).

- If \( \text{rm}(a) \in O_c(l, p) \): By assumption, \( l'_p \subset O_i(l, p) \in SeqPQ \). Therefore, \( a \) is in \( q'_j \) and is the should-be-removed value in \( O_c(l, p) \). Since each priority of values in \( O_c(l, p) \) is either larger or incomparable with priorities of values in \( O_c(l, p) \), we can removed \( a \) from \( q'_j \) and then \( q'_j \xrightarrow{a} q'_{j+1} \).

- If \( \text{rm}(a) \in O_i(l, p) \): By assumption we know that \( q_j \xrightarrow{\text{rm}(a)} q_{j+1} \). Since \( q'_j \) contains the same \( D_i(l, p) \) values as \( q_j \) and \( q'_j \) contains less \( D_c(l, p) \) values than \( q_j \), we can see that \( q'_j \xrightarrow{a} q'_{j+1} \).

For each \( q'_{\text{indj}} \), it is easy to see that \( q'_{\text{indj+1}} = q'_{\text{indj}} \). Therefore, we can see that \( l' = o_1, \ldots, o_{\text{indj-1}}, o_{\text{indj+1}}, \ldots \in SeqPQ \).

The following lemma shows that if a new sequence is generated by from some time point, erasing operations in \( O_i(l, p) \), then this new sequence is still in SeqPQ.

**Lemma 20.** Given a data-differentiated sequential execution \( l \in SeqPQ \) and a maximal priority \( p \) in \( l \), where \( l \) does not contain \( rm(\text{empty}) \). Let \( l' \) be generated from \( l \) by discarding operations in \( O_i(l, p) \) from some time point, then, \( l' \in SeqPQ \).

**Proof.** Let \( l = o_1, \ldots, o_m \), and \( q_0 \xrightarrow{a} q_1 \xrightarrow{a} \ldots \xrightarrow{a} q_m \) be the path of \( l \) in \( PQ \). Assume that \( l' \) is generated from \( l \) by discarding all operations \( o_i \) if \( 1 \leq i < O_i(l, p) \), and for a specific index \( k \). Let \( D \) be a set such that \( a \in D \), if \( put(a, \_ ) \) is in \( l \) and not in \( l' \). For each \( 0 \leq i \leq m \), let \( q'_i \) be generated from \( q_i \) by erasing values in \( D \).

Let \( l' = o'_1, \ldots, o'_j \), and let \( f \) be a function, such that \( f(i) = j \), if \( o'_j = o_j \).

- We can see that \( f \) maps each \( 0 \leq i \leq k-1 \) into \( i \), and \( q'_i \xrightarrow{a} q'_{i+1} \).
- It is easy to see that \( q'_{k-1} = q'_{f(k)-1} \), and for each \( i > k \), \( q'_{f(i)} = q'_{f(k+1)-1} \).

- If \( o_{f(k)} \) is a \( \text{put} \) operation, then it is obvious that \( q'_{f(k)-1} \xrightarrow{a} q'_{f(k)} \). Else, if \( o_{f(k)} = \text{rm}(a) \), we can see \( a \) is in \( q'_{f(k)-1} \), and since (1) \( q'_{f(k)-1} \) contains the same \( D_c(l, p) \) values as \( q_{f(k)-1} \) and \( q'_{f(k)-1} \) contains less \( D_i(l, p) \) values than \( q_{f(k)-1} \), and (2) each priority of values in \( O_i(l, p) \) is either larger or incomparable with priorities of values in \( O_c(l, p) \), we can see that \( q'_{f(k)-1} \xrightarrow{a} q'_{f(k)} \). Similarly we can prove the case of \( o_{f(j)} \) with \( j > k \).
XX:22 Checking Linearizability of Concurrent Priority Queues

This completes the proof of this lemma.

The following lemma shows that we can make \textit{put} with maximal priority to happen earlier.

\textbf{Lemma 21.} Given a data-differentiated sequential execution \(l \in \text{SeqPQ}\) and a maximal priority \(p\) in \(l\), where \(l\) does not contain \textit{rm(\text{empty})}. Let \(l = l_1 \cdot l_2\). Let \(l_3\) be the projection of \(l_2\) into \(\text{\{put(_-, p)\}}\), and \(l_4\) be the projection of \(l_2\) into other operations. Then, \(l' = l_1 \cdot l_3 \cdot l_4 \in \text{SeqPQ}\).

\textbf{Proof.} Let \(l = o_1 \cdots o_m\), and \(q_0 \xrightarrow{o_1} q_1 \xrightarrow{o_2} \cdots \xrightarrow{o_m} q_m\) be the path of \(l\) in \(PQ\). Let \(l_1 = o_1 \cdots o_n\), \(l_2 = o_{n+1} \cdots o_m\), let \(l_3 = o'_{n+1} \cdots o'_k\), let \(l_4 = o'_{k+1} \cdots o'_m\). Let \(f\) be a function, such that \(f(i) = j\), if \(o'_i = o_j\).

Let \(q'_i\) be constructed as follows:

- For \(0 \leq i \leq n\), let \(q'_i = q_i\).
- For \(n+1 \leq i \leq k\), let \(q'_i\) be obtained from \(q_n\) by adding values in \(o'_{n+1} \cdots o'_i\) with priority \(p\) and in the same order.
- For \(k+1 \leq i \leq m\), let \(q'_i\) be obtained from \(q_f(i)\) by adding values which are (1) with priority \(p\), (2) in \(o'_{n+1} \cdots o'_i\) and not removed by \(o_1 \cdots o_{f(i)}\). The order of adding them is the same as \(o'_{n+1} \cdots o'_i\).

Then, our proof proceeds as follows:

- It is obvious that \(q'_0 \xrightarrow{o_1} q'_1 \xrightarrow{o_2} \cdots \xrightarrow{o_m} q'_m\) and \(q'_n \xrightarrow{o_{n+1}} q'_{n+1} \xrightarrow{o_{n+2}} \cdots \xrightarrow{o_m} q'_m\).
- For \(q'_{k+1}\): We already know that \(q_f(k+1) \xrightarrow{o_f(k+1)} q_f(k+1)\), and it is easy to see that \(q_f(k+1)\) is obtained from \(q_n\) by adding values in \(o'_{n+1} \cdots o'_i\) and \(q'_{k+1}\) is obtained from \(q_f(k+1)\) by adding values which are (1) with priority \(p\), (2) in \(o'_{n+1} \cdots o'_i\) and not removed by \(o_1 \cdots o_{f(k+1)}\).

- If \(o_f(k+1)\) is an operation of non-\(p\) values, then \(q'_{k+1}\) is obtained from \(q_f(k+1)\) by adding values in \(o'_{n+1} \cdots o'_i\). Since non-\(p\) priority is either smaller or incomparable with \(p\), we can see that \(q'_k \xrightarrow{o_f(k+1)} q'_{k+1}\).
- Otherwise, it is only possible that \(o_f(k+1) = \text{rm}(a)\) for some value \(a\) with priority \(p\). We can see that \(q'_{k+1}\) is obtained from \(q_f(k+1)\) by adding values in \(o'_{n+1} \cdots o'_i\) and then remove \(a\). Since \(q_f(k+1) \xrightarrow{o_f(k+1)} q_f(k+1)\), in \(q_f(k+1)\) (and also in \(q'_k\)), there is no value with priority less than \(p\), and \(a\) is the first-input value of priority \(p\). Therefore, we can see that \(q'_k \xrightarrow{o_f(k+1)} q'_{k+1}\).

- For \(q'_{k+i}\) with \(i > 1\): We already know that \(q_f(k+i) \xrightarrow{o_f(k+i)} q_f(k+i)\), and \(q_f(k+i)\) is obtained from \(q_f(k+i-1)\) by adding values in \(o_f(k+i) \cdots o_f(k)\) and not removed by \(o_1 \cdots o_{f(k+i)}\).

We can see that \(q'_{k+i-1}\) (resp., \(q'_{k+i}\)) is obtained from \(q_f(k+i-1)\) (resp., \(q_f(k+i)\)) by adding values which are (1) with priority \(p\), (2) in \(o'_{n+1} \cdots o'_i\) and not removed by \(o_1 \cdots o_{f(k+i)}\) (resp., \(o_1 \cdots o_{f(k+i)}\)).

- If \(o_f(k+i)\) is an operation of non-\(p\) values, then \(q'_{k+i}\) is obtained from \(q_f(k+i)\) by adding values which are (1) with priority \(p\), (2) in \(o'_{n+1} \cdots o'_i\) and not removed by \(o_1 \cdots o_{f(k+i)}\). Since non-\(p\) priority is either smaller or incomparable with \(p\), we can see that \(q'_{k+i} \xrightarrow{o_f(k+i)} q'_{k+i}\).
Lemma 23. Given a data-differentiated sequential execution \( l \in \text{SeqPQ} \) and a maximal priority \( p \) in \( l \), where \( l \) does not contain \( \text{rm} \)(empty). Let \( l|_{O_c(l,p)} = l_1 \cdot l_2 \), let \( l' = l_1 \cdot l_3 \), where \( l_3 \) is the projection of \( l \) into non-\( l_1 \) operations. Then, \( l' \in \text{SeqPQ} \).

Proof. Let \( l = a_1 \cdot \ldots \cdot a_m \), and \( q_0 \overset{a_1}{\rightarrow} q_1 \cdot \ldots \cdot \overset{a_m}{\rightarrow} q_n \) be the path of \( l \) in \( \text{PQ} \). Let \( l_1 = o'_1 \cdot \ldots \cdot o'_{i-1} \), \( l_3 = o'_{i+1} \cdot \ldots \cdot o'_n \). Let \( f \) be a function, such that \( f(i) = j \), if \( o'_i = o_j \). Let \( D \) be the set of values which are added and not removed in \( o'_1 \cdot \ldots \cdot o'_n \).

Let \( q'_l \) be constructed as follows:

- It is easy to see that \( l_1 \in \text{SeqPQ} \), and let \( q'_0 \overset{a'_1}{\rightarrow} q'_1 \cdot \ldots \cdot \overset{a'_n}{\rightarrow} q'_n \) be the path of \( l_1 \) in \( \text{PQ} \).
- For \( n+1 \leq i \leq m \), let \( q'_i \) be obtained from \( q_f(i) \) by adding values in \( D \). The order of adding them is the same as \( o'_1 \cdot \ldots \cdot o'_n \).

Then, our proof proceeds as follows:

- We already know that \( q'_0 \overset{a'_1}{\rightarrow} q'_1 \cdot \ldots \cdot \overset{a'_n}{\rightarrow} q'_n \).
- For \( q'_{n+i} \): We already know that \( q_f(n+i-1) \overset{a_f(n+i)}{\rightarrow} q_f(n+i) \), and it is easy to see that \( q_f(n+i-1) \) is obtained from \( q_f(n+i-1) \) by adding \( D \)-values in \( q_f(n+i-1) \).

We can see that \( q'_{n+i} \) (resp., \( q'_{n+i} \)) is obtained from \( q_f(n+i-1) \) (resp., \( q_f(n+i) \)) by adding remaining values in \( D \).

- If \( a_f(n+i) \) is an operation of \( O_c(l, p) \) values, since priority in \( O_c(l, p) \) is either larger or incomparable with priority in \( O_c(l, p) \), we can see that \( q'_n \overset{a_f(n+1)}{\rightarrow} q'_{n+1} \).
- Otherwise, it is only possible that \( a_f(k+1) = \text{rm}(a) \) for some value \( a \) in \( O_c(l, p) \). Since \( q_f(n+i) \overset{a_f(n+i)}{\rightarrow} q_f(n+i) \), in \( q_f(n+i-1) \) (and also in \( q'_{n+i} \)), there is no value with priority less than \( p \), and \( a \) is the first-input value of priority \( p \). Therefore, we can see that \( q'_{n+i} \overset{a_f(n+i)}{\rightarrow} q'_{n+i} \).

This completes the proof of this lemma.

The following lemma shows that if a new sequence is generated by make some \( O_c(l, p) \) behaviors to happen earlier, then this new sequence is still in \( \text{SeqPQ} \).

Lemma 22. Given a data-differentiated sequential execution \( l \in \text{SeqPQ} \) and a maximal priority \( p \) in \( l \), where \( l \) does not contain \( \text{rm} \)(empty). Let \( l|_{O_c(l,p)} = l_1 \cdot l_2 \), let \( l' = l_1 \cdot l_3 \), where \( l_3 \) is the projection of \( l \) into non-\( l_1 \) operations. Then, \( l' \in \text{SeqPQ} \).

Proof. Let \( l = a_1 \cdot \ldots \cdot a_m \), and \( q_0 \overset{a_1}{\rightarrow} q_1 \cdot \ldots \cdot \overset{a_m}{\rightarrow} q_n \) be the path of \( l \) in \( \text{PQ} \). Let \( l_1 = o'_1 \cdot \ldots \cdot o'_{i-1} \), \( l_3 = o'_{i+1} \cdot \ldots \cdot o'_n \). Let \( f \) be a function, such that \( f(i) = j \), if \( o'_i = o_j \). Let \( D \) be the set of values which are added and not removed in \( o'_1 \cdot \ldots \cdot o'_n \).

Let \( q'_l \) be constructed as follows:

- It is easy to see that \( l_1 \in \text{SeqPQ} \), and let \( q'_0 \overset{a'_1}{\rightarrow} q'_1 \cdot \ldots \cdot \overset{a'_n}{\rightarrow} q'_n \) be the path of \( l_1 \) in \( \text{PQ} \).
- For \( n+1 \leq i \leq m \), let \( q'_i \) be obtained from \( q_f(i) \) by adding values in \( D \). The order of adding them is the same as \( o'_1 \cdot \ldots \cdot o'_n \).

Then, our proof proceeds as follows:

- We already know that \( q'_0 \overset{a'_1}{\rightarrow} q'_1 \cdot \ldots \cdot \overset{a'_n}{\rightarrow} q'_n \).
- For \( q'_{n+i} \): We already know that \( q_f(n+i-1) \overset{a_f(n+i)}{\rightarrow} q_f(n+i) \), and it is easy to see that \( q_f(n+i-1) \) is obtained from \( q_f(n+i-1) \) by adding \( D \)-values in \( q_f(n+i-1) \).

We can see that \( q'_{n+i} \) (resp., \( q'_{n+i} \)) is obtained from \( q_f(n+i-1) \) (resp., \( q_f(n+i) \)) by adding remaining values in \( D \).

- If \( a_f(n+i) \) is an operation of \( O_c(l, p) \) values, since priority in \( O_c(l, p) \) is either larger or incomparable with priority in \( O_c(l, p) \), we can see that \( q'_n \overset{a_f(n+1)}{\rightarrow} q'_{n+1} \).
- Otherwise, it is only possible that \( a_f(k+1) = \text{rm}(a) \) for some value \( a \) in \( O_c(l, p) \). Since \( q_f(n+i) \overset{a_f(n+i)}{\rightarrow} q_f(n+i) \), in \( q_f(n+i-1) \) (and also in \( q'_{n+i} \)), there is no value with priority less than \( p \), and \( a \) is the first-input value of priority \( p \). Therefore, we can see that \( q'_{n+i} \overset{a_f(n+i)}{\rightarrow} q'_{n+i} \).

This completes the proof of this lemma.

The following lemma shows that if a new sequence is generated by replacing a prefix with another one which make the priority queue has same content, then this new sequence is still in \( \text{SeqPQ} \).

Lemma 23. Given a data-differentiated sequential execution \( l \in \text{SeqPQ} \). Let \( l = l_1 \cdot l_2 \).

Given \( l_3 \in \text{SeqPQ} \). Assume that the priority queue has same content after executing \( l_1 \) and \( l_3 \). Let \( l' = l_3 \cdot l_2 \). Then, \( l' \in \text{SeqPQ} \).
Checking Linearizability of Concurrent Priority Queues

Proof. Let \( l = o_1 \ldots o_m \) and let \( q_0 \xrightarrow{o_1} q_1 \ldots \xrightarrow{o_m} q_m \) be the path of \( l \) in \( PQ \). Let \( l_1 = o_1 \ldots l_k \), \( l_3 = o'_1 \ldots o'_n \) and let \( q_0 \xrightarrow{o'_1} q_1' \ldots \xrightarrow{o'_n} q_m' \) be the path of \( l_3 \) in \( PQ \).

By assumption we know that \( q_k = q_m' \). Then it is not hard to see that \( q_0 \xrightarrow{o'_1} q_1' \ldots \xrightarrow{o'_n} q_m' \) is a path in \( PQ \), and then \( l' \in \text{SeqPQ} \). By Lemma 5 we know that \( l' \in \text{SeqPQ} \).

C.3 Proving Step-by-Step Linearizability of \( \text{SeqPQ} \)

With the help of Lemma 19, Lemma 20, Lemma 21, Lemma 22 and Lemma 23, we can now prove that \( \text{MatchedMaxPriority-Conc}(e, x) \) for some \( x \), then \( e \not\subseteq \text{SeqPQ} \Rightarrow e \not\subseteq \text{SeqPQ} \).

Proof. By assumption we know that there exists \( l \), such that \( e \subseteq l \) and \( \text{MatchedMaxPriority-Seq}(l, x) \) holds. Let \( p \) be the priority of \( x \), then \( l = u \cdot \text{put}(x, p) \cdot v \cdot \text{rm}(x) \cdot w \) for some \( u, v \) and \( w \). Let \( e' = e \setminus x \). By assumption there exists sequence \( l' \), such that \( e' \not\subseteq l' \in \text{SeqPQ} \). Let \( e_{lp} \) be an execution with linearization points of \( e \) and the linearization points is added according to \( l' \). Or we can say, \( e_{lp} \) is generated from \( e \) by instrumenting linearization points, and the projection of \( e_{lp} \) into operations is \( l' \). Let \( l'_v \) be the shortest prefix of \( l' \) that contains all operation of \( u \cdot v \).

Let \( U, V \) and \( W \) be the set of operations of \( u, v \) and \( w \), respectively. Let us change \( O_c(l, p) \) elements in \( U, V \) and \( W \), while keep \( O_c(l, p) \) elements unchanged. We proceed by a loop: In the first round of the loop, we start from the first \( O_c(l, p) \)-value of \( l'_v \), and let it be \( o_h \).

- Case 1: If in \( e_{lp} \), the linearization point of \( o_h \) is before \( \text{ret} (\text{rm}, x) \), and no \( O_c(l, p) \)-value in \( W \) happens before \( o_h \). Then, \( o_h \) is in the new version of \( U \cup V \).
- Case 2: Else, if in \( e_{lp} \), the linearization point of \( o_h \) is before \( \text{ret} (\text{rm}, x) \), and there exists \( O_c(l, p) \)-value \( o_w \) in \( W \), such that \( o_w <_{hb} o_h \). Then, in \( l'_v \), we put \( o_h \) and all \( O_c(l, p) \)-value whose linearization points is after the linearization point of \( o_h \) into new version of \( W \), and then stop the process of changing \( U \), \( V \) and \( W \).
- Case 3: Else, if in \( e_{lp} \), the linearization point of \( o_h \) is after \( \text{ret} (\text{rm}, x) \). Then, in \( l'_v \), we put \( o_h \) and all \( O_c(l, p) \)-value whose linearization points is after the linearization point of \( o_h \) into new version of \( W \), and then stop the process of changing \( U \), \( V \) and \( W \).

In the next round of the loop, we consider the second \( O_c(l, p) \)-value of \( l'_v \), and so on. Our process proceed, until either all element in \( l'_v \) are in new version of \( U \cup V \), or case 2 or case 3 happens and this process terminates. Let \( U' \cup V' \) and \( W' \) be the new version of \( U \cup V \) and \( W \) after the process terminates, respectively. Let \( O_+ \) be the set of operations that are moved into \( U' \cup V' \) in the process, and let \( O_- \) be the set of operations that are moved into \( W' \) in the process.

Let \( l'_{uv} \) be the projection of \( l' \) into \( U' \cup V' \), let \( O_x \) be the set of \( \text{put}(\_ , p) \) while the value is not \( x \) in \( h \). Let \( l'_1 \) be the longest prefix of \( l'_{uv} \), where linearization of each operation of \( l'_1 \) is before \( \text{ret} (\text{put}, b) \) in \( e_{lp} \). Let \( l'_2 \) be the projection of \( l' \) into operations of \( O_+ \) that are not in \( l'_1 \). Let \( l'_3 = l'_1 \cdot l'_2 \). Let \( l'_3 \) be the projection of \( l' \) into operations of \( l'_{uv} \), that are not in \( l'_1 \). Let \( l'_4 \) be the projection of \( l' \) into operations which are not in \( l'_{uv} \). Let \( l'' = l'_4 \cdot \text{put}(x, p) \cdot l''_2 \cdot \text{rm}(x) \cdot l''_3 \).

To prove \( e \not\subseteq l'' \), 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 one of the following case holds

- \( o_1 \) happens-before \( o_2 \) in \( h \),
- the operation corresponding to \( o_1 \) in \( l'' \) is before the one corresponding to \( o_2 \).
Assume there is a cycle in $G$. According the the property of interval order and the fact that the order of $l''$ is total, we know that there must exists $o_1$ and $o_2$, such that $o_1$ happens-before $o_2$ in $h$, but the corresponding operations are in the opposite order in $l''$. Then, we consider all possible case of $o_1$ and $o_2$ as follows: Let $O_a$ and $O_d$ be the set of operations in $l''_a$ and $l''_d$, respectively.

- If $o_2 \in l''_d \land o_1 \in l''_a$:
  - If $o_1, o_2 \in O_d$ or $o_1, o_2 \in O_a$: Then $l''$ contradicts with happen before relation of $h$.
  - If $o_2 \in O_a \land o_1 \in O_d$: Then the order of linearization points of $e_p$, contradicts with happen before relation of $h$.
- If $o_2 \in l''_d \land o_1 = put(x, p)$:
  - If $o_2 \in O_a$: This is impossible, since the linearization point of operations in $O_a$, is before $ret(put, x, p)$ in $e_p$.
  - If $o_2 \in O_d$: Then $l''$ contradicts with happen before relation of $h$.
- If $o_2 \in l''_d \land o_1 = rm(x)$:
  - If $o_2 \in O_a$: This violates the order of linearization point in $e_p$.
  - If $o_2 \in O_d$: According to $l''$, we can see that $put(x, p)$ does not happen before any operation in $O_a$. Then we can see that the linearization point of $o_1$ is before $ret(put, x, p)$ and $o_1 \in O_a$. This violates that $o_1 \in l''_a$.
- If $o_2 \in l''_d \land o_1 = l''_a$:
  - If $o_1 \in W \land o_2 \in U \cup V$: Then $l''$ contradicts with happen before relation of $h$.
  - If $o_1 \in W \land o_2 \in O_+$:
    - If $o_1 \in O_l(l, p)$: Then according to the construction process of $U' \cup V'$ and $W'$, we can see that $o_1 \in O_+$ and then $o_1 \in U' \cup V'$, which contradicts that $o_1 \in l''_w$.
    - If $o_1 \in O_c(l, p)$: Then according to the construction process of $U' \cup V'$ and $W'$, we can see that $o_2 \in O_-$, which contradicts that $o_2 \in O_+$.
  - If $o_1 \in O_- \land o_2 \in U \cup V$:
    - If the reason of $o_1 \in O_-$ is case 2: Let $o_h$ be as in case 2. Then there exists $O_c(l, p)$-value $o_w \in W$, and in $e_p$, $ret(o_w)$, the linearization point of $o_h$, is before the linearization point of $o_1$, and $ret(o_1)$ is before $call(o_2)$. Therefore, we can see that $o_w <_h o_2$, and then $l$ contradicts with happen before relation of $h$.
    - If the reason of $o_1 \in O_-$ is case 3: Let $o_h$ be as in case 3. Then in $e_p$, $ret(rm, x)$ is before the linearization point of $o_h$, the linearization point of $o_h$ is before the linearization point of $o_1$, and $ret(o_1)$ is before $call(o_2)$. Therefore, we can see that $rm(x) <_h o_2$, and then $l$ contradicts with happen before relation of $h$.
  - If $o_1 \in O_- \land o_2 \in O_+$: This is impossible, since in $e_p$, the linearization points of operations in $O_+$, is before the linearization points of operations in $O_-$.}

- If $o_2 = put(x, p) \land o_1 \in l''_d$: This is impossible, since in $e_p$, the linearization points of operations in $l''_d$ is after $ret(put, x, p)$.
- If $o_2 = put(x, p) \land o_1 = rm(x)$: Then $l$ contradicts with happen before relation of $h$.
- If $o_2 = put(x, p) \land o_1 \in l''_w$: This is impossible, since in $e_p$, the linearization points of operations in $l''_w$ is after $ret(put, x, p)$.
- If $o_1 \in W$: Then $l$ contradicts with happen before relation of $h$.
- If $o_1 \in O_-$:
Checking Linearizability of Concurrent Priority Queues

- If the reason of \( o_1 \in O_- \) is case 2: Let \( o_h \) be as in case 2. Then there exists \( O_c(l, p) \)-value \( o_w \in W \), and in \( e_{lp} \), \( ret(o_w) \) is before \( call(o_h) \), the linearization point of \( o_h \) is before the linearization point of \( o_1 \), and \( ret(o_1) \) is before \( call(put(x, p)) \). Therefore, we can see that \( o_w <_{hb} put(x, p) \), and then \( l \) contradicts with happen before relation of \( h \).

- If the reason of \( o_1 \in O_- \) is case 3: Let \( o_h \) be as in case 3. Then in \( e_{lp} \), \( ret(put(x, p)) \) is before the linearization point of \( o_h \), the linearization point of \( o_h \) is before the linearization point of \( o_1 \), and \( ret(o_1) \) is before \( call(put(x, p)) \). Therefore, we can see that \( rm(x) <_{hb} put(x, p) \), and then \( l \) contradicts with happen before relation of \( h \).

- If \( o_2 \in l''_1 \land o_1 \in l''_2 \): Then \( l' \) contradicts with happen before relation of \( h \).

- If \( o_2 \in l''_2 \land o_1 = \text{rm}(x) \): We can prove this similarly as the case of \( o_2 \in l''_1 \land o_1 = \text{rm}(x) \).

- If \( o_2 \in l''_2 \land o_1 \in l''_3 \): We can prove this similarly as the case of \( o_2 \in l''_1 \land o_1 \in l''_3 \).

- If \( o_2 = \text{rm}(x) \land o_1 \in l''_3 \):
  - If \( o_1 \in W \): Then \( l \) contradicts with happen before relation of \( h \).
  - If \( o_1 \in O_- \):
    - If the reason of \( o_1 \in O_- \) is case 2: Let \( o_h \) be as in case 2. Then there exists \( O_c(l, p) \)-value \( o_w \in W \), and in \( e_{lp} \), \( ret(o_w) \) is before \( call(o_h) \), the linearization point of \( o_h \) is before the linearization point of \( o_1 \).
      Since \( l \) is consistent with the happen before order of \( h \), we can see that \( call(\text{rm}, x) \) is before \( ret(o_w) \). Therefore, we can see that the linearization point of \( o_1 \) is after \( call(\text{rm}, x) \), and then it is impossible that \( o_1 <_{hb} \text{rm}(x) \).
    - If the reason of \( o_1 \in O_- \) is case 3: Let \( o_h \) be as in case 3. Then in \( e_{lp} \), \( ret(\text{rm}, x) \) is before the linearization point of \( o_h \), and the linearization point of \( o_h \) is before the linearization point of \( o_1 \). Therefore, we can see that the linearization point of \( o_1 \) is after \( ret(\text{rm}, x) \), and then it is impossible that \( o_1 <_{hb} \text{rm}(x) \).

- If \( o_2 \in l''_1 \land o_1 \in l''_3 \): Then \( l' \) contradicts with happen before relation of \( h \).

Therefore, we know that \( G \) is acyclic, and then we know that \( h \subseteq l'' \).

It remains to prove that \( l'' \in \text{SeqPQ} \). The process for proving \( l'' \in \text{SeqPQ} \) is as follows:

- Since \( l' \in \text{SeqPQ} \) and \( l'' \) is a prefix of \( l' \), it is obvious that \( l'' \in \text{SeqPQ} \).
- \( l''_u \) can be obtained from \( l'' \) as follows:
  - Discard \( O_c(l, p) \)-value that are in \( W \) and keep \( O_c(l, p) \)-value in \( U \cup V \) unchanged.
  - From some time point, discard all the \( O_i(l, p) \) operations after this time point.
  - From \( \text{matched}_{\omega}(u, v, p) \), we can see that \( O_c(l, p) \)-value in \( U \cdot V \) is matched, and then it is easy to see that the projection of \( l''_u \) in to \( O_c(l, p) \)-value in \( U \cdot V \) is still in \( \text{SeqPQ} \). By Lemma 19 and Lemma 20, we can see that \( l''_u \in \text{SeqPQ} \).
- \( l''_u \cdot l''_2 \) can be obtained from \( l''_u \) as follows: Execute until reaching some time point \( t \), then first execute all \( O_x \) operations after \( t \), and then execute remaining operations. By Lemma 21, we can see that \( l''_u \cdot l''_2 \in \text{SeqPQ} \).
- Let \( l''_1 \) be obtained from \( l' \) by discarding \( O_c(l, p) \)-value in \( U \cdot V \). By Lemma 3, we can see that \( l''_1 \in \text{SeqPQ} \).
- Let \( l''_1|O_i(l, p) = l''_j \cdot o_f \), where \( l''_j \) is the projection of \( O_i(l, p) \)-value in \( U \cdot V \). Let \( l''_h \) be obtained from \( l''_1 \) by discarding operations in \( l''_j \). By Lemma 22, we can see that \( l''_j \cdot l''_h \in \text{SeqPQ} \).
- By Lemma 3, it is obvious that \( l''_j \in \text{SeqPQ} \). From \( \text{matched}_{\omega}(u, v, p) \), we can see that the content of priority queue after executing \( l''_j \cdot l''_2 \) is the same as after executing \( l''_j \). By Lemma 23, we can see that \( l''_j \cdot l''_2 \cdot l''_h \in \text{SeqPQ} \). It is easy to see that \( l''_h = l''_2 \), and then \( l''_j \cdot l''_2 \cdot l''_h \in \text{SeqPQ} \).
- Since \( \text{MatchedMaxPriority-Seq}(l, x) \) holds, it is easy to see that \( l'' = l''_j \cdot \text{put}(x, p) \cdot l''_2 \cdot \text{rm}(x) \cdot l''_h \in \text{SeqPQ} \).
Lemma 25. For each data-differentiated execution $e$ where $\text{UnmatchedMaxPriority-Conc}(e, x)$ for some $x$, then $e \setminus x \subseteq \text{SeqPQ} \Rightarrow e \subseteq \text{SeqPQ}$.

Proof. By assumption we know that there exists $l$, such that $e \subseteq l$ and $\text{UnmatchedMaxPriority-Seq}(l, x)$ holds. Let $lp$ be the priority of $e$ and the linearization points is added according to $l'$. Or we can say, $lp$ is generated from $e$ by instrumenting linearization points, and the projection of $lp$ into operations is $l'$.

Let $l''$ be the shortest prefix of $l'$ such that linearization point of each operation of $l''$ is before ret($put, x, p$) in $lp$. Let $Oa$ be the set of $put(\_, p)$ while the value is not in $h$. Let $l''_a$ be the projection of $l''$ into operations of $Oa$ that are not in $l''$. Let $l''_2 = l''_1 \cdot l''_2$. Let $l''_2$ be the projection of $l''$ into operations of $l'$ that are not in $l''$. Let $l''_3 = l''_1 \cdot \text{put}(x, p) \cdot l''_2$. To prove $h \sqsubseteq l''$, we define graph $G$ as in Lemma 24. Assume that there is a cycle in $G$, then there must exists $o_1$ and $o_2$, such that $o_1$ happens-before $o_2$ in $h$, but the corresponding operations are in the opposite order in $l''$. Then, we consider all possible case of $o_1$ and $o_2$ as follows: Let $Oa$ and $Ox$ be the set of operations in $l''_1$ and $l''_2$, respectively.

- If $o_2 \in l''_1 \land o_1 \in l''_1$:
  - If $o_1, o_2 \in Oa$ or $o_1, o_2 \in Ox$: Then $l'$ contradicts with happen before relation of $h$.
  - If $o_2 \in Oa \land o_1 \in Ox$: It is not hard to see that $put(x, p) <_{hb} o_2$. Then, it is impossible to locate the linearization point of $o_2$ before ret($put, x, p$) in $lp$.

- If $o_2 \in l''_1 \land o_1 = put(x, p)$:
  - If $o_2 \in Oa$: This is impossible, since in $lp$, the linearization point of $o_1$ is before ret($put, x, p$).
  - If $o_2 \in Ox$: This is impossible, since $o_2 = put(\_, p)$, and $l$ is consistent with happen before relation of $h$.

- If $o_2 \in l''_1 \land o_1 \in l''_2$:
  - If $o_2 \in Oa$: This is impossible, since in $lp$, the linearization point of operation in $l''_2$ is before the linearization point of operations in $l''_2$.
  - If $o_2 \in Ox$: Since no put(\_, p) happens before put($x, p$) in $h$, call($o_2$) is before ret($put, x, p$).
    Since $o_1 <_{hb} o_2$, we can see that ret($o_1$) is before call($o_2$), and then ret($o_1$) is before ret($put, x, p$). Then the linearization point of $o_1$ can only be before ret($put, x, p$), and $o_1 \in l''_2$, which contradicts that $o_1 \in l''_2$.

- If $o_2 = put(x, p) \land o_1 \in l''_2$: Then since the linearization point of $o_1$ can only be before ret($put, x, p$), we can see that $o_1 \in l''_2$, which contradicts that $o_1 \in l''_2$.

- If $o_2 \in l''_2 \land o_1 \in l''_2$: Then $l'$ contradicts with happen before relation of $h$.

Therefore, we know that $G$ is acyclic, and then we know that $e \subseteq l''$.

It remains to prove that $l'' \subseteq \text{SeqPQ}$, $l''_1 \cdot l''_2 \cdot l''_3$ can be obtained from $l'$ as follows: Execute until reaching some time point $t$, then first execute all $Oa$ operations after $t$, and then execute remaining operations. By Lemma 24, we can see that $l''_1 \cdot l''_2 = l''_1 \cdot l''_2 \cdot l''_3 \in \text{SeqPQ}$. Since UnmatchedMaxPriority-Seq($l, x$) holds, it is easy to see that $l'' = l''_1 \cdot \text{put}(x, p) \cdot l''_2 \in \text{SeqPQ}$.

Therefore, we prove that $e \subseteq l'' \subseteq \text{SeqPQ}$. This completes the proof of this lemma.

Lemma 26. For each data-differentiated execution $e$ where EmptyRemove-Conc($e, x$) for some $o = \text{rm}(\text{empty})$, then $e \setminus o \subseteq \text{SeqPQ} \Rightarrow e \subseteq \text{SeqPQ}$.
Checking Linearizability of Concurrent Priority Queues

**Proof.** By assumption we know that there exists \( l \), such that \( e \sqsubseteq l \) and EmptyRemove-Seq\((l,o)\) holds. Then \( l = u \cdot o \cdot v \) for some \( u \) and \( v \). Let \( e' = e \setminus o \). By assumption there exists sequence \( l' \), such that \( e' \sqsubseteq l' \in \text{SeqPQ} \).

Let \( E_L \) be the set of operations in \( u \) and \( E_R \) be the set of operations in \( v \). Let \( l'_L = l'|_{E_L} \) and \( l'_R = l'|_{E_R} \). Since priority queue is closed under projection (Lemma 3) and all the put operations and \( rm \) in \( u \) are matched, we know that \( l'_L \in \text{SeqPQ} \) and the the priority queue is empty after executing \( l'_L \). Then we know that \( l'_L \cdot \text{rm}(\text{empty}) \in \text{SeqPQ} \). Since \( l'_R \) is obtained from \( l' \) by discarding pairs of matched put and \( rm \) operations, it is easy to see that \( l'_R \in \text{SeqPQ} \), and then we know that \( l'' = l'_L \cdot o \cdot l'_R \in \text{SeqPQ} \).

It remains to prove that \( h \sqsubseteq l'' \). To prove \( h \sqsubseteq l'' \), we define graph \( G \) as in Lemma 24. Assume that there is a cycle in \( G \), then there must exists \( o_1 \) and \( o_2 \), such that \( o_1 \) happens-before \( o_2 \) in \( h \), but the corresponding operations are in the opposite order in \( l'' \). Then, we consider all possible case of \( o_1 \) and \( o_2 \) as follows:

- \( o_1, o_2 \in l'_L \), or \( o_1, o_2 \in l'_R \): Then \( l' \) contradicts with happen before relation of \( h \).
- If \( o_1 = o \wedge o_2 \in l'_L \), or \( o_1 \in l'_R \wedge o_2 \in l'_L \), or \( o_1 \in l'_R \wedge o_2 = o \), then \( l \) contradicts with happen before relation of \( h \).

Therefore, we know that \( G \) is acyclic, and then we know that \( h \sqsubseteq \text{SeqPQ} \).

The following lemma states that \( \text{SeqPQ} \) is step-by-step linearizability, which is a direct consequence of Lemma 24, Lemma 25, and Lemma 26.

**Lemma 27.** \( \text{SeqPQ} \) is step-by-step linearizability.

**Proof.** This is a direct consequence of Lemma 24, Lemma 25, and Lemma 26.

C.4 Proof of Lemma 6

**Lemma 6** \( \text{Check-PQ-Conc}(e) = \text{true} \) iff \( e \sqsubseteq \text{SeqPQ} \), for every data-differentiated execution \( e \).

**Proof.** To prove the if direction, given a data-differentiated \( e \sqsubseteq l \in \text{SeqPQ} \). Then we being a loop as follows,

- If Has-EmptyRemoves\((e)\) holds, then \( l = u \cdot o \cdot v \) for some \( o = \text{rm}(\text{empty}) \). It is not hard to see that EmptyRemove-Seq\((l,o)\) holds. Let \( e' \) be the projection of \( e \) into operations of \( u \cdot v \). It is easy to see that \( e' \sqsubseteq u \cdot v \) and then \( e' \sqsubseteq \text{SeqPQ} \). Then we start the next round and choose \( e' \) to be the “\( e \) in the next round”.
- If Has-UnmatchedMaxPriority\((e)\) holds, then there exists a maximal priority \( p \) in \( e \), such that \( p \) has unmatched \( \text{put} \). Let \( l = u \cdot \text{put}(x,p) \cdot v \), where in \( v \) there is no \( \text{put} \) with priority \( p \). It is not hard to see that UnmatchedMaxPriority-Seq\((l,x)\) holds. Let \( e' \) be the projection of \( e \) into operations of \( u \cdot v \). It is easy to see that \( e' \sqsubseteq u \cdot v \) and then \( e' \sqsubseteq \text{SeqPQ} \). Then we start the next round and choose \( e' \) to be the “\( e \) in the next round”.
- If Has-MatchedMaxPriority\((e)\) holds, then there exists a maximal priority \( p \) in \( e \), such that \( p \) has only matched \( \text{put} \). Let \( l = u \cdot l' \cdot \text{put}(x,p) \cdot v \cdot \text{rm}(x) \cdot w \), where in \( v \cdot w \) there is no \( \text{put} \) with priority \( p \). It is not hard to see that MatchedMaxPriority-Seq\((l,x)\) holds. Let \( e' \) be the projection of \( e \) into operations of \( u \cdot v \cdot w \). It is easy to see that \( e' \sqsubseteq u \cdot v \cdot w \) and then \( e' \sqsubseteq \text{SeqPQ} \). Then we start the next round and choose \( e' \) to be the “\( e \) in the next round”.

The loop terminates when at some round \( e = e' \). This process ensures that \( \text{Check-PQ-Conc}(e) = \text{true} \).
To prove the only if direction, given a data-differentiated execution $e$ and assume that $Check-PQ-Conc(e) = \text{true}$. Then we have $e \overset{\Gamma}{\rightarrow} e_1 \ldots \overset{\Gamma}{\rightarrow} e_m$ with $e_0 = e$ and $e_m = \epsilon$. Assume that for each $i$, we found $\Gamma-Seq(e_i, \alpha_i)$ holds for some $\alpha_i$. From $\Gamma-\text{Seq}(e_m, l_m)$ holds and $e_m = e_{m-1} \setminus \alpha_{m-1} \subseteq \text{SeqPQ}$, by Lemma 27 we can see that $e_{m-1} \subseteq \text{SeqPQ}$. Similarly, we can prove that $e_{m-2}, \ldots, e_0 = e \subseteq \text{SeqPQ}$.

### C.5 Proof of Lemma 7

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

**Proof.** To prove the only if direction, assume that $e \subseteq l \in \text{SeqPQ}$. Given $e' = e|_D$ and $l' = l|_D$, it is easy to see that $e' \subseteq l'$, and by Lemma 6 we can see that $l' \subseteq \text{SeqPQ}$. Then it is obvious that $Check-PQ-Conc-NonRec(e')$ returns true.

To prove the if direction, assume that for each $e' \in \text{proj}(e)$, $Check-PQ-Conc-NonRec(e')$ returns true. Then similarly as by Lemma 6 we can implies that $Check-PQ-Conc(e) = \text{true}$, and then by Lemma 8 we know that $e \subseteq \text{SeqPQ}$.

### D Proofs and Definitions in Section 4

To facilitate our proof, we consider two cases of MatchedMaxPriority-Seq($e, x$) depending on whether $e$ contains exactly one value with priority $p$ or at least two values. We denote by MatchedMaxPriority$^+$($e, x$) the strengthening of MatchedMaxPriority($e, x$) with the condition that all the values other than $x$ have a priority strictly smaller than $p$, and by MatchedMaxPriority$^-$($e, x$) the strengthening of the same formula with the negation of this condition.

Similarly, we consider two cases of UnmatchedMaxPriority-Seq($e, x$) depending on whether $e$ some values of priority $p$ has matched put. We denote by UnmatchedMaxPriority$^+$($e, x$) the strengthening of UnmatchedMaxPriority($e, x$) with the condition that non value of priority $p$ has matched put, and by UnmatchedMaxPriority$^-$($e, x$) the strengthening of the same formula with the negation of this condition.

### D.1 Lemma and Register Automata For FIFO of Single-Priority Executions

Before we go to investigate $\Gamma$-linearizable automata, we can use the result in [5] to simplify our work. [5] states that checking linearizability w.r.t queue can be reduced into checking emptiness of intersection between $\mathcal{I}$ and a set of automata. Given a data-differentiated execution $e$, let $e|_i$ be an execution generated from $e$ by erasing call and return actions of values that does not use priority $i$ (does not influence $rm(\text{empty})$). We call a extended priority queue execution with only one priority a single-priority execution. Let $\text{transToQueue}(e)$ be an execution generated from $e$ by transforming put and rm into enq and deq, respectively, and then discarding priorities. We can see that for each $e \in \text{SeqPQ}$ and each priority $i$, $\text{transToQueue}(e|_i)$ satisfy FIFO (first in first out) property.

Given an execution of queue, we say that it is differentiated, if each value is enqueued at most once. [5] states that, given a differentiated queue execution $e$ without $deq(\text{empty})$, $e$ is not linearizable with respect to queue, if one of the following cases holds for some $a, b$: (1) $deq(b) <_h \text{enq}(b)$, (2) there are no $\text{enq}(b)$ and at least one $deq(b)$, (3) there are one $\text{enq}(b)$ and more than one $deq(b)$, and (4) $\text{enq}(a) <_h \text{enq}(b)$, and $deq(b) <_h \text{deq}(a)$, or $\text{deq}(a)$ does not exists. For each such case, we can construct a register automata for extended priority queue.
We generate register automata $A_{\text{SinPri}}^1$ for the first case, and it is shown in Figure 8. Here $C_1 = \{\text{call}(\text{put}, a, \text{true}), \text{ret}(\text{put}, a, \text{true}), \text{call}(\text{rm}, a), \text{ret}(\text{rm}, a), \text{call}(\text{rm}, \text{empty}), \text{ret}(\text{rm}, \text{empty})\}$, $C_2 = C_1 \cup \{\text{ret}(\text{rm}, b)\}$, $C_3 = C_2 \cup \{\text{ret}(\text{put}, b, \text{true})\}$.

![Figure 8 Automaton $A_{\text{SinPri}}^1$](image)

We generate register automata $A_{\text{SinPri}}^2$ for the second case, and it is shown in Figure 9. Here $C_1 = \{\text{call}(\text{put}, a, \text{true}), \text{ret}(\text{put}, a, \text{true}), \text{call}(\text{rm}, a), \text{ret}(\text{rm}, a), \text{call}(\text{rm}, \text{empty}), \text{ret}(\text{rm}, \text{empty})\}$, $C_2 = C_1 \cup \{\text{call}(\text{rm}, b) + \text{ret}(\text{rm}, b)\}$.

![Figure 9 Automaton $A_{\text{SinPri}}^2$](image)

We generate register automata $A_{\text{SinPri}}^3$ for the third case, and it is shown in Figure 10. Here $C_1 = \{\text{call}(\text{put}, a, \text{true}), \text{ret}(\text{put}, a, \text{true}), \text{call}(\text{rm}, a), \text{ret}(\text{rm}, a), \text{call}(\text{rm}, \text{empty}), \text{ret}(\text{rm}, \text{empty})\}$, $C_2 = C_1 \cup \{\text{ret}(\text{put}, b, \text{true})\}$, $C_3 = C_2 \cup \{\text{ret}(\text{rm}, b)\}$, $C_4 = C_3 \cup \{\text{call}(\text{rm}, b)\}$, $C_5 = C_4 \cup \{\text{ret}(\text{rm}, b)\}$, $C_6 = C_5 \cup \{\text{call}(\text{rm}, b)\}$.

![Figure 10 Automaton $A_{\text{SinPri}}^3$](image)

We generate register automata $A_{\text{SinPri}}^4$ for the forth case, and it is shown in Figure 11. Here $C_1 = C \cup \{\text{call}(\text{rm}, b)\}$, and $C_2 = C \cup \{\text{ret}(\text{put}, b, = \text{r}), \text{call}(\text{rm}, a), \text{ret}(\text{rm}, a)\}$, where $C = \{\text{call}(\text{put}, d, \text{true}), \text{ret}(\text{put}, d, \text{true}), \text{call}(\text{rm}, d), \text{ret}(\text{rm}, d), \text{call}(\text{rm}, \text{empty}), \text{ret}(\text{rm}, \text{empty})\}$.

Let $A_{\text{SinPri}}$ be the union of $A_{\text{SinPri}}^1, A_{\text{SinPri}}^2, A_{\text{SinPri}}^3, A_{\text{SinPri}}^4$. Let us prove Lemma 28.

Lemma 28. Given a data-independent implementations $I$ of extended priority queue $\mathcal{I}$ and $A_{\text{SinPri}} \neq \emptyset$, if and only if there exists $e \in I_{\neq}$, $e' \in \text{proj}(e)$, such that $e'$ is single-priority without $\text{rm}($empty$)$, and $\text{transToQueue}(e')$ does not linearizable to queue.
Proof. [5] states that, given a differentiated queue execution $e$ without $\text{deq}(\text{empty})$, $e$ is not linearizable with respect to queue, if one of the following cases holds for some $v_a, v_b$: (1) $\text{deq}(v_b) <_{hb} \text{enq}(v_a)$, (2) there are are no $\text{enq}(v_b)$ and at least one $\text{deq}(v_b)$, (3) there are one $\text{enq}(v_b)$ and more than one $\text{deq}(v_b)$, and (4) $\text{enq}(v_a) <_{hb} \text{enq}(v_b)$, and $\text{deq}(v_b) <_{hb} \text{deq}(v_a)$, or $\text{deq}(v_a)$ does not exists.

Let us prove the only if direction. Assume that there exists execution $e_0 \in \mathcal{I}$ and $e_0$ is accepted by $A_{\text{SimPri}}^1, A_{\text{SimPri}}^2, A_{\text{SimPri}}^3$ or $A_{\text{SimPri}}^4$. By data-independence, we can see that there exists a data-differentiated $e \in \mathcal{I}$ and renaming function, such that $e_0 = r(e)$. Let $e'$ be obtained from $e$ by first removing $\text{rm}(\text{empty})$, and then,

- If $e_0$ is accepted by $A_{\text{SimPri}}^1, A_{\text{SimPri}}^2$ or $A_{\text{SimPri}}^3$: Then remove all values that are not renamed into $b$ by $r$.
- If $e_0$ is accepted by $A_{\text{SimPri}}^4$: Then remove all values that are not renamed into $a$ or $b$ by $r$.

It is obvious that $e' \in \text{proj}(e)$. It is easy to see that $\text{transToQueue}(e')$ satisfies one of above conditions, and then $\text{transToQueue}(e')$ is not linearizable w.r.t queue.

Let us prove the if direction. Assume that exists $e \in \mathcal{I}, e' \in \text{proj}(e)$, such that $e'$ is single-priority without $\text{rm}(\text{empty})$, and $\text{transToQueue}(e')$ does not linearizable to queue. Then we construct a renaming function $r$ as follows:

- If this is because case 1, case 2 or case 3: $r$ maps $v_b$ into $b$ and maps all other values into $a$.
- If this is because case 4: $r$ maps $v_a$ and $v_b$ into $a$ and $b$, respectively, and maps all other values into $d$.

Then it is not hard to see that $r(e) \in \mathcal{I}$ and it is accepted by $A_{\text{SimPri}}^1, A_{\text{SimPri}}^2, A_{\text{SimPri}}^3$ or $A_{\text{SimPri}}^4$. This completes the proof of this lemma.

### D.2 Proof of Lemma 10

The following lemma states that, from linearization of sub-histories, we can merge them and obtain a linearization (regardless of whether it belongs to sequential specification) of the whole history.

**Lemma 29.** Given an execution $e$, operation sets $S_1, S_2$ and sequences $l_1$ and $l_2$. Let $e_1 = e|S_1$ and $e_2 = e|S_2$. Assume that $e_1 \subseteq l_1, e_2 \subseteq l_2$, and $S_1 \cup S_2$ contains all operations of $e$. Then, there exists a sequence $l$, such that $e \subseteq l, l|S_1 = l_1$ and $l|S_2 = l_2$.

**Proof.** Given an execution $e$ and a operation $o \in S_2$, let $MB(o) = \{ o' | o' \in S_1 \text{ and } o' \text{ happens before } o \text{ in } h \}$, let $\text{SBI}(o) = \min\{i | l_1[i, i] \text{ contains all elements of } MB(o)\}$.

Let $l = s_1 \cdot l_2[1] \cdot \ldots \cdot l_2[n] \cdot s_{n+1}$ be generated as follows, where $n = |l_2|$: [Diagram showing execution sequence]
XX:32 Checking Linearizability of Concurrent Priority Queues

- $s_1 = l_1[0, SBI[l_2[1]]]$, 
- If $s_1 \cdot l_2[1] \cdot \ldots \cdot l_2[i]$ already contains $l_1(SBI[l_2[i+1]])$, then $s_{i+1} = e$. Otherwise, $s_{i+1}$ is a subsequence of $l_1$, which starts from the next of last elements of $s_1 \cdot l_2[1] \cdot \ldots \cdot l_2[i]$ in $l_1$ and ends in $l_1(SBI[l_2[i+1]])$.

It is obvious that $l_1S_1 = l_1$ and $l_1S_2 = l_2$, and it remains to prove that $e \sqsubseteq l$. We prove this by contradiction. Assume that $e \not\sqsubseteq l$. Then there must be two operations $o_1, o_2$ of $e$, such that $o_1 <_{hb} o_2$ in $e$ but $o_2$ before $o_1$ in $l$. Since $l_1S_1 = l_1, l_1S_2 = l_2$, and $h_1 \sqsubseteq l_1, h_2 \sqsubseteq l_2$, it is easy to see that it is impossible that $o_1, o_2 \in S_1$ or $o_1, o_2 \in S_2$. There are only two possibilities:

- $o_1 \in S_1 \land o_2 \in S_2$. Then we can see that $o_2 = l_2[i]$ and $o_1 \in s_j$ for some $i < j$. Since $o_1 <_{hb} o_2$, we know that $o_1 \in SBI(o_2)$. By the construction of $l$, we know that $o_1$ must be in $s_k$ for some $k < i$, contradicts that $o_1 \in s_j$ with $i < j$.
- $o_1 \in S_2 \land o_2 \in S_1$. Then we can see that $o_2 \in s_i$ and $o_1 = l_2[j]$ for some $i < j$. It is easy to see that this leads to contradiction when $i = j$. For the case of $i \neq j$, we need to satisfy the following requirements: (1) $o_1 (l_2[j])$ does not happen before $l_2[i]$, (2) $l_2[i]$ does not happen before $o_2$, (3) $o_2$ is either overlap or happens before $o' \in MB(l_2[i])$, and (4) $o' <_{hb} l_2[i]$. By enumeration we can see that it is impossible that above four conditions be satisfied while $o_1 <_{hb} o_2$.

This completes the proof of this lemma.

With Lemma[29] we can now prove Lemma[10].

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

**Proof.** We deal with the case of $\Gamma = \text{MatchedMaxPriority}$, and other cases can be similarly dealt with.

To prove the only if direction, assume that $e \sqsubseteq l$, MatchedMaxPriority-Seq($l, x$) holds and $p$ is the priority of $x$. We can see that $e \sqsubseteq u \cdot \text{put}(x, p) \cdot v \cdot \text{rm}(x) \cdot w$. Let $u', v'$ and $w'$ be obtained from $u, v$ and $w$ by erasing all values with priority incomparable with $p$, respectively. Since the predicates in MatchedMaxPriority-Seq($l, x$) does not restrict the values of priorities incomparable with $p$, it is not hard to see that $e \sqsubseteq l'$ and MatchedMaxPriority-Seq($l', x$) holds, where $l' = u' \cdot \text{put}(x, prl) \cdot v' \cdot \text{rm}(x) \cdot w'$.

To prove the if direction, given $e' = e|_{<p}$. By assumption, $e' \sqsubseteq l$, MatchedMaxPriority-Seq($l, x$) holds and $p$ is the priority of $x$, and then we can see that $e' \sqsubseteq l_1 = u \cdot \text{put}(x, p) \cdot v \cdot \text{rm}(x) \cdot w$. Let $O_e$ be the set of operations in $e$ that have priorities comparable with $p$, and let $O_i$ be the set of operations in $e$ that have priorities incomparable with $p$. It is obvious that $l_1$ is the linearization of $e|_{O_e}$. By Lemma [29] there exists sequence $l$, such that $e \sqsubseteq l$, and $l|_{O_e} = l_1$. Then $l = u' \cdot \text{put}(x, p) \cdot v' \cdot \text{rm}(x) \cdot w'$, where $u'|_{O_e} = u, v'|_{O_e} = v$ and $w'|_{O_e} = w$. Since $p$ is one of maximal priorities in $e$, and the predicates in MatchedMaxPriority-Seq($l, x$) does not restrict the values with priority $O_i$, it is easy to see that MatchedMaxPriority-Seq($l, x$) holds.

Therefore, from now on, it is safe to consider only the data-differentiated sequences with only one maximal priority.
D.3 Proofs, Definitions and Register Automata in Subsection 4.1.1

Let us introduce $UVSet(e, x)$, which intuitively contains all pairs of operations that should be putted before $rm(x)$ when construction linearization of $x$ according to MatchedMaxPriority. Let $UVSet_1(e, x) = \emptyset$ or either $o < hput(x, \_)$ or $rm(x)$, or there exists $o'$ with the same value of $o$, such that $o' < hput(x)$ or $rm(x)$}. For each $i > 1$, let $UVSet_{i+1}(e, x) = \emptyset$ or $UVSet_k$ for each $k \leq i$, and either $o$ happens before some operation $o' \in UVSet_i(e, x)$, or there exists $o''$ with the same value of $o$, and $o''$ happens before some operation $o' \in UVSet_i(e, x)$. Let $UVSet(e, x) = UVSet_i(e, x) \cup \ldots$. Note that it is possible that $UVSet_i(e, x) \cap UVSet_j(e, x) = \emptyset$ for any $i \neq j$.

The following lemma states that $UVSet(e, x)$ contains only matched put and rm.

\begin{itemize}
  \item \textbf{Lemma 30.} 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 $x$. Assume that $G$ has no cycle going through $x$. Then, $UVSet(e, x)$ contains only matched put and rm.
  \end{itemize}

\textbf{Proof.} We prove this lemma by contradiction. Assume that there exists a value, such that $UVSet(e, x)$ contains only its put and does not contain its rm. Then we can see that there exists $d_1, \ldots, d_j$. Intuitively, $d_1, \ldots, d_j$ are elements in $UVSet_1(e, x), \ldots, UVSet_i(e, x)$, respectively. $UVSet(e, x)$ contains $put(d_{i-1}, \_)$ and does not contain $rm(d_j)$. And each $d_i$ is the reason of $d_{i+1} \in UVSet_{i+1}(e, x)$. Formally, we require that

- For each $1 \leq i \leq j$, operations of $d_i$ belongs to $UVSet_i(e, x)$.
- For each $i \neq j$, $put(d_{i-1}, \_), rm(d_i) \in UVSet_i(e, x)$, $put(d_{j-1}, \_) \in UVSet_j(e, x)$, and $e$ does not contain $rm(d_j)$.
- An operation of $d_1$ happens before an operations of $x$. For each $1 < i \leq j$, an operation of $d_i$ happens an operation of $d_{i-1}$.
- For each $k$ and $\text{ind}$, if $k > \text{ind} + 1$, then no operation of $d_k$ happens before operation of $d_{\text{ind}}$.

According to the definition of $UVSet(e, x)$, it is easy to see that such $d_1, \ldots, d_j$ exists. Let us prove the following fact:

\textbf{fact}_1: Given $1 \leq i < j$, it can not be the case that $put(d_{i-1}, \_)$ and $rm(d_j)$ overlap.

\textbf{Proof of fact}_1: We prove fact_1 by contradiction. Assume that for some $i \neq j$, $put(d_{i-1}, \_)$ and $rm(d_j)$ overlap. Since $put(d_{i-1}, \_), rm(d_i) \in UVSet_i(h, x)$, we know that an operation $o_i$ of $d_i$ happens before operation $o_{i-1}$ of $d_{i-1}$. Moreover, since $put(d_{i-1}, \_)$ and $rm(d_i)$ overlap, it is not hard to see that the call action of $put(d_{i-1}, \_)$ and the call action of $rm(d_i)$ is before the call action of $o_{i-1}$. Since operations of $d_{i-1}$ is in $UVSet_{i-1}(e, x)$, we know that an operation $o_{i-1}'$ of $d_{i-1}$ happens before operation $o_i'$ of $d_i$. Then, it is not hard to see that $o_{i-1}'$ also happens before $o_{i-1}$, which contradicts that for each $k > \text{ind} + 1$, no operation of $d_k$ happens before operation of $d_{\text{ind}}$.

We already know that an operation of $d_1$ happens before an operation of $x$. By fact_1, we can ensure that $put(d_{i-1}, \_)$ happens before an operation of $x$, and then $d_1 \to x$ in $G$. For each $1 < i \leq j$, we know that an operation $o_i$ of $d_i$ happens before an operation $o_{i-1}$ of $d_{i-1}$. By fact_1, we can ensure that $o_i = put(d_{i-1}, \_)$ and $o_{i-1} = rm(d_i)$, and then $d_1 \to d_{i-1}$ in $G$. Since $h$ contains $put(d_{j-1}, \_)$ and does not contain $rm(d_j)$, we know that $x \to d_j$ in $G$. Then $G$ has a cycle going through $x$, which contradicts that $G$ has no cycle going through $x$.

The following lemma states that $UVSet(e, x)$ does not happen before $rm(x)$ when the left-right constraint has no cycle going through $x$.

\begin{itemize}
  \item \textbf{Lemma 31.} 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...
Checking Linearizability of Concurrent Priority Queues

Let us consider all the possible cases of \((t_1, t_2, t_3)\):

- \((\text{put}, \text{put}, \text{put})\): Then \(rm(x) <_{hb} \text{put}(d_{j+1}) <_{hb} \text{put}(x, p)\), contradicts that \(rm(x)\) does not happen before \(\text{put}(x, p)\).
- \((\text{put}, \text{put}, \text{rm})\): Then \(rm(x) <_{hb} \text{put}(d_{j+1}) <_{hb} \text{rm}(x)\), contradicts that \(rm(x)\) does not happen before \(\text{rm}(x)\).
- \((\text{put}, \text{rm}, \text{put})\): Then \((rm(x) <_{hb} \text{put}(d_{j+1})) \land (rm(d) <_{hb} \text{put}(x, p))\). By interval order, we know that \((rm(x) <_{hb} \text{put}(x, p)) \lor (rm(d) <_{hb} \text{put}(d_{j+1}))\), which is impossible.
- \((\text{put}, \text{rm}, \text{rm})\): Then \((rm(x) <_{hb} \text{put}(d_{j+1})) \land (rm(d) <_{hb} \text{rm}(x))\). We can see that \(rm(d) <_{hb} \text{rm}(x)\), which contradicts that \(rm(d)\) does not happen before \(\text{put}(d_{j+1})\).
- \((\text{rm}, \text{put}, \text{put})\): Then \((rm(x) <_{hb} \text{rm}(d)) \land (\text{put}(d_{j+1}) <_{hb} \text{rm}(x))\). We can see that \(x, d\) has circle in \(G\), contradicts that \(G\) has no cycle going through \(x\).
- \((\text{rm}, \text{put}, \text{rm})\): Then \((rm(x) <_{hb} \text{rm}(d)) \land (\text{put}(d_{j+1}) <_{hb} \text{rm}(x))\). We can see that \(x, d\) has circle in \(G\), contradicts that \(G\) has no cycle going through \(x\).
- \((\text{rm}, \text{rm}, \text{put})\): Then \(rm(x) <_{hb} \text{rm}(d) <_{hb} \text{put}(x, p)\), contradicts that \(rm(x)\) does not happen before \(\text{put}(x, p)\).
- \((\text{rm}, \text{rm}, \text{rm})\): Then \(rm(x) <_{hb} \text{rm}(d) <_{hb} \text{rm}(x)\), contradicts that \(rm(x)\) does not happen before \(\text{rm}(x)\).

This completes the proof for \(UVSet_1(e, x)\).

Assume we already prove that for some \(j \geq 1\), \(rm(x)\) does not happen before any operation in \(UVSet_1(e, x) \cup \ldots \cup UVSet_j(e, x)\). Let us prove that \(rm(x)\) does not happen before any operation in \(UVSet_{j+1}(e, x)\) by contradiction. Assume that \(rm(x) <_{hb} o\), where \(o \in UVSet_{j+1}(e, x)\). We use a triple \((t_1, t_2, t_3)\) to represent related information. \(t_1, t_2, t_3\) are chosen from \(\{\text{put}, \text{rm}\}\). \(t_1\) represents whether \(o\) is a \text{put} operation or a \text{rm} operation. \(t_2\) and \(t_3\) is used for the reason of \(o \in UVSet_1(e, x)\): \(o \in UVSet_1(e, x)\), since an operation (of kind \(t_2\)) of \(d\) happens before an operation (of kind \(t_3\)) of \(x\). Let us consider all the possible cases of \((t_1, t_2, t_3)\):

- \((\text{put}, \text{put}, \text{put})\): Then \(rm(x) <_{hb} \text{put}(d_{j+1}) <_{hb} \text{put}(d_{j+1})\). We can see that \((rm(x) <_{hb} \text{put}(d_{j+1}) \land (\text{put}(d_{j+1}) \in UVSet_j(e, x)))\), which contradicts that \(rm(x)\) does not happen before any operation in \(UVSet_1(e, x) \cup \ldots \cup UVSet_j(e, x)\).
- \((\text{put}, \text{put}, \text{rm})\): Then \(rm(x) <_{hb} \text{put}(d_{j+1}) <_{hb} \text{rm}(d_{j+1})\). We can see that \((rm(x) <_{hb} \text{rm}(d_{j+1}) \land (\text{rm}(d_{j+1}) \in UVSet_j(e, x)))\), which contradicts that \(rm(x)\) does not happen before any operation in \(UVSet_1(e, x) \cup \ldots \cup UVSet_j(e, x)\).
- \((\text{put}, \text{rm}, \text{put})\): Then \((rm(x) <_{hb} \text{put}(d_{j+1}) \land (rm(d_{j+1}) <_{hb} \text{put}(d_{j+1}))\) By interval order, we know that \((rm(x) <_{hb} \text{put}(d_{j+1}) \lor (rm(d_{j+1}) <_{hb} \text{put}(d_{j+1}))\), which is impossible.
- (put, rm, rm): Then \( (rm(x) <_{hb} put(d_{j+1}, _{-})) \land \neg (rm(d_{j+1}) <_{hb} rm(d_j)) \). By interval order, we know that \( (rm(x) <_{hb} rm(d_j)) \lor (rm(d_{j+1}) <_{hb} put(d_{j+1}, _{-})) \), which is impossible.

- (rm, put, put): Then \( (rm(x) <_{hb} rm(d_{j+1})) \land \neg (put(d_{j+1}, _{-}) <_{hb} put(d_j, _{-})) \). Let us consider the reason of \( put(d_{j+1}, _{-}) \in UVSet_j(e, x) \):
  - If \( (j > 1) \land (put(d_{j-1}) <_{hb} o") \), where \( o" \) is an operation of value \( d_{j-1} \) and \( put(d_{j-1}, _{-}) \), \( rm(d_{j-1}) \) \( \in UVSet_{j-1}(e, x) \): Then since \( (put(d_{j+1}, _{-}) <_{hb} put(d_j, _{-})) \land (put(d_j, _{-}) <_{hb} o") \), we can see that \( put(d_{j+1}, _{-}) <_{hb} o" \), and then operations of \( d_{j+1} \) in is in \( UVSet_j(e, x) \), contradicts that operations of \( d_{j+1} \) is in \( UVSet_j(e, x) \).
  - If \( (j > 1) \land (rm(d_j) <_{hb} o") \), where \( o" \) is an operation of \( x \): Similar to above case.
  - If \( (j > 1) \land \neg (rm(d_j) <_{hb} o") \), where \( o" \) is an operation of value \( d_j \) and \( put(d_{j-1}, _{-}) \), \( rm(d_{j-1}) \) \( \in UVSet_{j-1}(e, x) \): Then since \( (put(d_{j+1}, _{-}) <_{hb} put(d_j, _{-})) \land (rm(d_j) <_{hb} o") \), we can see that \( (put(d_{j+1}, _{-}) <_{hb} o") \lor (rm(d_j) <_{hb} put(d_j, _{-})) \), which is impossible.

- (rm, rm, put): Let \( T_{ind} \) be the set of sentences \( \{ rm(x) <_{hb} rm(d_{j+1}), put(d_{j+1}, _{-}) <_{hb} rm(d_j), \ldots, put(d_{ind+1}, _{-}) <_{hb} rm(d_{ind}) \} \). Here each \( d_i \) is a value of some operation in \( UVSet_1(e, x) \). Let us prove that from \( T_j \) we can obtain contradiction by induction:

**Base case 1:** From \( T_1 \) we can obtain contradiction.

Let us prove base case 1:

- If \( put(d_{1}, _{-}) \) happens \( o \), and \( o \) is an operation of \( x \). Then there is a cycle \( x \rightarrow d_{j+1} \rightarrow \ldots \rightarrow d_{1} \rightarrow x \) in \( G \), contradicts that \( G \) has no cycle going through \( x \).
- If \( rm(d_1) \) happens before \( o \), and \( o \) is an operation of \( x \). Then since \( put(d_{2, _{-}}) <_{hb} rm(d_1) \) and \( rm(d_1) <_{hb} o \), we can see that \( put(d_{2, _{-}}) <_{hb} o \), and then \( put(d_{2, _{-}}) \in UVSet_1(e, x) \), contradicts that \( put(d_{2, _{-}}) \in UVSet_2(e, x) \).

**Base case 2:** From \( T_2 \) we can obtain contradiction.

Let us prove base case 2: If \( rm(d_2) <_{hb} o \), and \( o \) is an operation of \( d_2 \), then since \( put(d_{3, _{-}}) <_{hb} rm(d_2) \) and \( rm(d_2) <_{hb} o \), we know that \( put(d_{3, _{-}}) <_{hb} o \). This implies that \( put(d_{3, _{-}}) \in UVSet_2(e, x) \), contradicts that \( rm(d_{3, _{-}}) \in UVSet_3(e, x) \). Therefore, it is only possible that \( put(d_{2, _{-}}) \) happens before an operation of \( d_1 \).

- If \( put(d_{2, _{-}}) <_{hb} put(d_{1, _{-}}) \) and \( put(d_{1, _{-}}) \) happens before operations of \( x \), then we know that \( put(d_{2, _{-}}) \) happens before operation of \( x \), which is impossible.
- If \( put(d_{2, _{-}}) <_{hb} put(d_{1, _{-}}) \) and \( rm(d_1) \) happens before operations of \( x \), then by interval order, we know that \( put(d_{2, _{-}}) \) happens before operation of \( x \), or \( rm(d_1) <_{hb} put(d_{1, _{-}}) \), which is impossible.
- If \( put(d_{2, _{-}}) <_{hb} rm(d_1) \) and \( put(d_{1, _{-}}) \) happens before operations of \( x \), then \( x \rightarrow d_{j+1} \rightarrow \ldots \rightarrow d_1 \rightarrow x \) in \( G \), contradicts that \( G \) has no cycle going through \( x \).
- If \( put(d_{2, _{-}}) <_{hb} rm(d_1) \) and \( rm(d_1) \) happens before operations of \( x \), then we know that \( put(d_{2, _{-}}) \) happens before operation of \( x \), which is impossible.

**induction step:** Given \( ind \geq 3 \), if from \( T_{ind, 1} \) we can obtain contradiction, then from \( T_{ind} \) we can also contain contradiction.

Prove of the induction step: Similarly as base case 2, we can prove that it is only possible that \( put(d_{ind, _{-}}) \) happens before operations of \( d_{ind-1} \).

- If \( put(d_{ind, _{-}}) <_{hb} put(d_{ind-1, _{-}}) \) and \( put(d_{ind-1, _{-}}) \) happens before operations of \( d_{ind-2} \), then we know that \( put(d_{ind}) \) happens before operation of \( d_{ind-2} \), which is impossible.
Checking Linearizability of Concurrent Priority Queues

- If \( put(d_{i,\_}) <_{hb} put(d_{i+1,\_}) \) and \( rm(d_{i+1}) \) happens before operations of \( d_{i\_2} \), then by interval order, we know that \( put(d_{i,\_}) \) happens before operation of \( d_{i\_2} \), or \( rm(d_{i\_1}) <_{hb} put(d_{i+1,\_}) \), which is impossible.
- If \( put(d_{i,\_}) <_{hb} rm(d_{i\_1}) \), then we obtain \( T_{i\_1} \), which already contain contradiction.

By base case 1, base case 2 and the induction step, it is easy to see that for each \( i \), \( T_i \) contains contradiction. Therefore, \( T \), the case of \( rm, put, rm \), contains contradiction.

- \( (rm, rm, put) \): Then \( (rm(x) <_{hb} rm(d_{j+1})) \) \& \( (rm(d_{j+1}) <_{hb} put(d_{j\_1})) \). We can see that \( (rm(x) <_{hb} put(d_{j\_1})) \) \& \( (put(d_{j\_1}) \in UVSet_j(e, x)) \), which contradicts that \( rm(x) \) does not happen before any operation in \( UVSet_1(e, x) \cup \ldots \cup UVSet_j(e, x) \).

- \( (rm, rm, rm) \): Then \( (rm(x) <_{hb} rm(d_{j+1})) \)& \( (rm(d_{j+1}) <_{hb} rm(d_{j})) \). We can see that \( (rm(x) <_{hb} rm(d_j)) \) \& \( (rm(d_j) \in UVSet_j(e, x)) \), which contradicts that \( rm(x) \) does not happen before any operation in \( UVSet_1(e, x) \cup \ldots \cup UVSet_j(e, x) \).

This completes the proof for \( UVSet_{j+1}(e, x) \). Therefore, \( rm(x) \) does not happen before any operation in \( UVSet_1(e, x) = UVSet_1(e, x) \cup UVSet_2(e, x) \cup \ldots \).

With Lemma \([30]\) and Lemma \([31]\) we can now prove Lemma \([13]\) which give the equivalent characterization of non MatchedMaxPriority-linearizable executions.

**Lemma** \([13]\) Given a data-differentiated execution \( e \) where Has-MatchedMaxPriority(o) 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 \( x \).

**Proof.** To prove the only if direction, assume that \( e \not\subseteq l \) and MatchedMaxPriority-Seq(l, x) holds. Then \( l = u \cdot put(x, p) \cdot v \cdot rm(x) \cdot w \), and let \( U, V \) and \( W \) be the set of operations of \( u, v \) and \( w \), respectively. Assume by contradiction that, there is a cycle \( d_1 \rightarrow d_2 \rightarrow \ldots \rightarrow d_m \rightarrow x \rightarrow d_1 \) in \( G \). It is obvious that the priority of each \( d_i \) is smaller than \( p \). Then our proof proceeds as follows:

According to the definition of left-right constraint, there are two possibilities. The first possibility is that, \( rm(x) \) happens before \( rm(d_{1}) \). It is obvious that \( rm(d_{1}) \in W \), and then since \( U \cup V \) contains matched \( put \) and \( rm \), we can see that \( put(d_{1}) \), \( rm(d_{1}) \in W \). Then,

- Since \( d_1 \rightarrow d_2 \), by definition of \( G \), we know that \( put(d_1) \) happens before \( rm(d_2) \). Since \( put(d_{1}) \in W \) and \( U \cup V \) contains matched \( put \) and \( rm \), we know that \( put(d_2) \), \( rm(d_2) \in W \). Similarly, for each \( 1 \leq i \leq m \), we know that \( put(d_{i}) \), \( rm(d_{i}) \in W \).

- Since \( d_m \rightarrow x \),
  - if \( put(d_m) \) happens before \( put(x) \), then we can see that \( put(d_m) \in U \), which contradicts that \( put(d_m) \in W \).
  - if \( put(d_m) \) happens before \( rm(x) \), then we can see that \( put(d_m) \in U \cup V \), which contradicts that \( put(d_m) \in W \).

The second possibility is that, \( e \) contains one \( put(d_{1\_1}) \) and no \( rm(d_{1\_1}) \). Note that for each \( j > 1 \), \( e \) contains \( put(d_{j\_1}) \) and \( rm(d_j) \). Since \( d_m \rightarrow x \), is is obvious that \( put(d_m) \in U \cup V \). Since \( U \cup V \) contains matched \( put \) and \( rm \), we know that \( put(d_m) \), \( rm(d_m) \in U \cup V \). Then, since \( d_{m\_1} \rightarrow d_m \), by definition of \( G \), we know that \( put(d_{m\_1}) \) happens before \( rm(d_m) \). Since \( rm(d_m) \in U \cup V \) and \( U \cup V \) contains matched \( put \) and \( rm \), we know that \( put(d_{m\_1}) \), \( rm(d_{m\_1}) \in U \cup V \). Similarly, for each \( 1 \leq i \leq m \), we know that \( put(d_{i\_1}) \), \( rm(d_i) \in U \cup V \), and also \( put(d_1) \in U \cup V \). However, there is one \( put(d_{1\_1}) \) and no \( rm(d_1) \) in \( e \), contradicts that \( U \cup V \) contains matched \( put \) and \( rm \).

This completes the proof of the only if direction.
To prove the if direction, assume that $G$ has no cycle going through $x$. Let $E_u$ be the set of operations that happen before $\text{put}(x)$ in $e$. It is easy to see that $E_u \subseteq \text{UVSet}(e, x)$. Let $E_v = \text{UVSet}(e, x) \setminus E_u$. Let $E_o$ be the set of operations of $e$, and let $E_w = E_o \setminus \text{UVSet}(e, x)$.

By Lemma 33, we can see that $E_u \cup E_o$ contains matched $\text{put}$ and $\text{rm}$ operations. It remains to prove that for $E_u$, $\{\text{put}(x, p)\}$, $E_o$, $\{\text{rm}(x)\}$, $E_w$, no elements of the latter set happens before elements of the former set. We prove this by showing that all the following cases are impossible:

- Case 1: Some operation $o_w \in E_w$ happens before $\text{rm}(x)$. Then we know that $o_w \in \text{UVSet}(e, x) = E_u \cup E_o$, which contradicts that $o_w \in E_w$.
- Case 2: Some operation $o_w \in E_w$ happens before some operation $o_w \in E_u \cup E_v$. Then we know that $o_w \in \text{UVSet}(e, x) = E_u \cup E_v$, which contradicts that $o_w \in E_w$.
- Case 3: Some operation $o_w \in E_w$ happens before $\text{put}(x)$. Then we know that $o_w \in \text{UVSet}(e, x) = E_u \cup E_v$, which contradicts that $o_w \in E_w$.
- Case 4: $\text{rm}(x)$ happens before some $o_w \in \text{UVSet}(e, x) = E_u \cup E_o$. By Lemma 31, we know that this is impossible.
- Case 5: $\text{rm}(x)$ happens before $\text{put}(x)$. This contradicts that each single-priority projection satisfy the FIFO property.
- Case 6: Some operation $o_v \in E_v$ happens before $\text{put}(x)$. Then we know that $o_v \in E_u$, which contradicts that $o_v \in E_v$.
- Case 7: Some operation $o_v \in E_v$ happens before some operation $o_w \in E_u$. Then we know that $o_v \in E_v$, which contradicts that $o_v \in E_u$.
- Case 8: $\text{put}(x)$ happens before some operation $o_w \in E_u$. This is impossible.

This completes the proof of the if direction.

Let us begin to represent register automata that is used for capture the existence of a data-differentiated execution $e$, let $p$ be a maximal priority of $e$, and $e'$ be the projection of $e$ into values with priorities comparable with $p$. We also require that Has-MatchedMaxPriority($e'$) holds and there exists a cycle going through the value with maximal priority in $e'$. By data-independence, we can obtain $e_r$ from $e$ by renaming function, which maps such value to be $b$, maps values that cover it to be $a$, and maps other values into $d$. There are four possible enumeration of call and return actions of $\text{put}(b, p)$ and $\text{rm}(b)$. For each of them, we generate a register automaton.

For the case when $e_v|_b = \text{call}(\text{put}(b, p)) \cdot \text{ret}(\text{put}(b, p)) \cdot \text{call}(\text{rm}(b)) \cdot \text{ret}(\text{rm}(b))$, we generate register automaton $A_{\text{lar}}^1$, as shown in Figure 12. Here $C_1 = C \cup \{\text{ret}(\text{rm}(a))\}$, $C_2 = C \cup \{\text{call}(\text{put}(a, < r))\}$, $C_3 = C_2 \cup \{\text{ret}(\text{rm}(a))\}$, where $C = \{\text{call}(\text{put}, \top, \text{true}), \text{ret}(\text{put}, \top, \text{true}), \text{call}(\text{rm}, \top), \text{ret}(\text{rm}, \top), \text{call}(\text{rm}, \text{empty}), \text{ret}(\text{rm}, \text{empty})\}$. The differentiated branch in $A_{\text{lar}}^1$ comes from the positions of the first $\text{ret}(\text{put}, a, \_)$.

$A_{\text{lar}}^1$ is used to recognize conditions in Figure 13. Here for simplicity, we only draw operation of $b$, and the first $\text{ret}(\text{put}, a, \_)$.

For the case when $e_v|_b = \text{call}(\text{put}(b, p)) \cdot \text{call}(\text{rm}(b)) \cdot \text{ret}(\text{put}(b, p)) \cdot \text{ret}(\text{rm}(b))$, we generate register automaton $A_{\text{lar}}^2$, as shown in Figure 14. Here $C_1, C_2, C_3$ is the same as that in $A_{\text{lar}}^1$. The differentiated branch in $A_{\text{lar}}^2$ comes from the positions of the first $\text{ret}(\text{put}, a, \_)$.

$A_{\text{lar}}^2$ is used to recognize conditions in Figure 15. Here for simplicity, we only draw operation of $b$, and the first $\text{ret}(\text{put}, a, \_)$.

For the case when $e_v|_b = \text{call}(\text{rm}(b)) \cdot \text{call}(\text{put}(b, p)) \cdot \text{ret}(\text{put}(b, p)) \cdot \text{ret}(\text{rm}(b))$, we generate register automaton $A_{\text{lar}}^3$, as shown in Figure 16. Here $C_1, C_2, C_3$ is the same as that in $A_{\text{lar}}^1$. The differentiated branch in $A_{\text{lar}}^3$ comes from the positions of the first $\text{ret}(\text{put}, a, \_)$.
Checking Linearizability of Concurrent Priority Queues

Figure 12 Automaton $A_{1\text{-lar}}$

Figure 13 Conditions recognized by $A_{1\text{-lar}}$

Figure 14 Automaton $A_{2\text{-lar}}$

Figure 15 Conditions recognized by $A_{2\text{-lar}}$
\[ q_{\text{init}} r = \ast \]

\[ q_2 C_2 q_8 C_2 q_{10} C_3 \]

\[ q_3 C_2 \]

\[ q_4 C_3 \]

\[ q_5 C_3 \]

\[ q_6 C_4 \]

\[ q_7 \]

**Figure 16** Automaton \( A_3^{\text{lar}} \)

\[ \begin{align*}
& \text{put}(b, p) \\
& \text{rm}(b) \\
& \text{put}(a, \_)
\end{align*} \]

\[ e_1 \]

**Figure 17** Conditions recognized by \( A_3^{\text{lar}} \)

\( A_3^{\text{lar}} \) is used to recognize conditions in Figure 17. Here for simplicity, we only draw operation of \( b \), and the first \( \text{ret}(\text{put}, a, \_). \)

For the case when \( e_r|_b = \text{call}(\text{rm}, b) \cdot \text{call}(\text{put}, b, p) \cdot \text{ret}(\text{rm}, b) \cdot \text{ret}(\text{put}, b, p) \), we generate register automaton \( A_4^{\text{lar}} \), as shown in Figure 18. Here \( C_1, C_2, C_3 \) is the same as that in \( A_3^{\text{lar}} \), and \( C_4 = C_1 \cup \{ \text{ret}(\text{put}, b, = r) \} \). The differentiated branch in \( A_4^{\text{lar}} \) comes from the positions of the first \( \text{ret}(\text{put}, a, \_). \)

**Figure 18** Automaton \( A_4^{\text{lar}} \)

\[ \begin{align*}
& \text{put}(b, p) \\
& \text{rm}(b) \\
& \text{put}(a, \_)
\end{align*} \]

\[ e_2 \]

\[ q_{\text{init}} r = \ast \]

\[ q_2 C_2 q_8 C_2 q_{10} C_3 \]

\[ q_3 C_2 \]

\[ q_4 C_3 \]

\[ q_5 C_3 \]

\[ q_6 C_4 \]

\[ q_7 \]

**Figure 19** Automaton \( A_4^{\text{lar}} \)

\( A_4^{\text{lar}} \) is used to recognize conditions in Figure 19. Here for simplicity, we only draw operation of \( b \), and the first \( \text{ret}(\text{put}, a, \_). \)

Let \( A_{1 \text{-lar}} \) be the union of \( A_1^{\text{lar}}, A_2^{\text{lar}}, A_3^{\text{lar}} \) and \( A_4^{\text{lar}} \). The following lemma states that \( A_{1 \text{-lar}} \) is MatchedMaxPriority^\sim-complete.

**Lemma 32.** \( A_{1 \text{-lar}} \) is MatchedMaxPriority^\sim-complete.

**Proof.** We need to prove that, given a data-independent implementation \( I \), \( A_{1 \text{-lar}} \cap I \neq \emptyset \) if and
only if there exists \( e \in I \) and \( e' \in \text{proj}(e) \) such that \( e' \) is not MatchedMaxPriority\(^>\)-linearizable.

By Lemma 10 and Lemma 13, we need to prove the following fact:

**Fact:** Given a data-independent implementation \( \mathcal{I} \), \( \mathcal{A}_{1,\text{lar}} \cap \mathcal{I} \neq \emptyset \) if and only if there exists \( e \in I \) and \( e' \in \text{proj}(e) \), Has-MatchedMaxPriority\(^>\)(\( e' \)) holds, \( x \) is the value with maximal priority in \( e' \), \( e' \) has only one maximal priority, and there is a cycle going through \( x \) in \( G \), where \( G \) is the left-right constraint of \( x \) in \( e' \).

The **only if** direction: Let us consider the case of \( \mathcal{A}_{1,\text{lar}} \). Assume that \( e_1 \in I \) is accepted by \( \mathcal{A}_{1,\text{lar}} \). By data-independence, there exists data-differentiated execution \( e \in I \) and renaming function \( r_1 \), such that \( e_1 = r_1(e) \). Assume that \( r_1 \) maps \( d \) into \( b \) and maps \( f_1, \ldots, f_m \) into \( a \). Let \( e' \) be obtained from \( e \) by projection into \( \{d, f_1, \ldots, f_m\} \). Assume that the priority of \( b \) is \( p \). It is easy to see that \( e' \) has only one maximal priority, Has-MatchedMaxPriority\(^>\)(\( e' \)) holds, and there is a cycle going through \( d \) in \( G \), where \( G \) is the left-right constraint of \( e' \). The case of \( \mathcal{A}_{2,\text{lar}}, \mathcal{A}_{3,\text{lar}} \) and \( \mathcal{A}_{4,\text{lar}} \) can be similarly proved.

The **if** direction: Given such \( e \), \( e' \) and \( x \). Let renaming function \( r \) maps \( x \) into \( b \), maps values cover \( x \) into \( a \), and maps other values into \( d \). By data-independence, \( r(e) \in I \). Then depending on the cases of \( r(e) \), we can see that \( r(e) \) is accepted by \( \mathcal{A}_{1,\text{lar}}, \mathcal{A}_{2,\text{lar}}, \mathcal{A}_{3,\text{lar}} \) or \( \mathcal{A}_{4,\text{lar}} \).

---

**D.4 Proofs, Definitions and Register Automata in Subsection 4.1.2**

Let \( \text{Items}(e, p) \) be the set of values with priority \( p \) in execution \( e \). The following lemma states a method to build the linearization of \( e \) w.r.t MatchedMaxPriority\(^\leq\).

**Lemma 33.** Given a data-differentiated execution \( e \) where Has-MatchedMaxPriority\(^\leq\)(\( e \)) holds and \( p \) is its maximal priority. If there exists a value \( x \) with priority \( p \), such that for each \( y \in \text{Items}(e, p) \), (1) \( x \not\approx_{pb} y \), and (2) the right-most gap-point of \( x \) is after \( \text{call}(\text{put}, y, p) \) and \( \text{call}(\text{rm}, y) \), then \( e \) is MatchedMaxPriority\(^\leq\)-linearizable.

**Proof.** Let \( o \) be the right-most gap-point of \( x \). We locate linearization points of each operation as follows:

- Locate the linearization point of \( \text{rm}(x) \) at \( o \).
- If \( \text{put}(x, p) \) overlaps with \( \text{rm}(x) \), then locate the linearization point of \( \text{put}(x, p) \) just before the linearization point of \( \text{rm}(x) \). Otherwise, \( \text{put}(x, p) <_{bh} \text{rm}(x) \), and we locate the linearization point of \( \text{put}(x, p) \) just before its return action.
- Locate linearization points of operation of each \( y \in \text{Items}(e, p) \) (except for \( x \)) just after the call action of the operation.
- For value \( z \) with priority smaller than \( p \). If both \( \text{call}(\text{put}, z, \_ ) \) and \( \text{call}(\text{rm}, z) \) is before \( o \), then locate the linearization points of \( \text{put}(z, \_) \) and \( \text{rm}(z) \) just after their call actions. If both \( \text{ret}(\text{put}, z) \) and \( \text{ret}(\text{rm}, z) \) (if exists) is after \( o \), then locate the linearization points of \( \text{put}(z, \_) \) and
rm(z) just before their return actions. Otherwise, x is in interval of z, which contradicts the definition of gap-point, and is impossible.

Let l be the sequence of linearization points constructed above. It is obvious that e ⊑ l. Since for each y ∈ Items(e, p), o is after call(put, y, _) and call(rm, x), we can see that rm(x) is after put(y, p) and rm(y) in l. It is obvious that put(x, p) is before rm(x) in l. Since x does not <pb to y, we can see that no put(y, p) happens before put(x, p). Then it is easy to see that put(x, p) is after put(y, p) in l. Since Has-MatchedMaxPriority=e(e) holds, all other values in Items(e, p) has matched put and rm, and it is easy to see that their put and rm (except for that of x) are all before rm(x) in l.

For value z with priority smaller than p, we can see that there are only two possibilities: (1) put(z, _) and rm(z) are both before rm(x) in l, and (2) put(z, _) and rm(z) (if exists) are after before rm(x) in l. Therefore, before rm(x) in l, the put and rm of z are matched.

Therefore, it is easy to see that MatchedMaxPriority=e-Seq(l, x) holds and e is MatchedMaxPriority=e-linearizable.

With Lemma[33] we can prove the following lemma, which states that getting rid of case in Figure[3] is enough for ensure that MatchedMaxPriority=e(e) holds.

Lemma[15] Let e be a data-differentiated execution which contains only one maximal priority p such that Has-MatchedMaxPriority(e) holds. Then, e is not linearizable w.r.t MatchedMaxPriority=e if 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).

Proof. To prove the if direction, let e*x,y be the execution that is obtained from e by erasing all actions of values that has same priority as x, except for actions of x and y. It is obvious that Has-MatchedMaxPriority=−priority=e(x,y) holds. Since y <pb x, we can see that x should be chosen as α in MatchedMaxPriority=−e.

According to Lemma[13] (Here we temporarily forget the existence of y), the only possible position for locating linearization point of rm(x) is at gap-point of x. Otherwise, if the linearization point of rm(x) is chosen at a position that is not a gap-point of x, then there exists unmatched operation before rm(x) with smaller priority. Since the rightmost gap-point of x is before call(put, y, p) or call(rm, y), if we locate linearization point of rm(x) at gap-point of x, then rm(x) will be before call(put, y, p) or call(rm, x).

Therefore, for every sequence l = u · put(x, p) · v · rm(x) · w, if e*x,y ⊑ l, then either u · v contains some unmatched operations of priority smaller than p, or w contains put(y, p) or rm(y). In both cases, MatchedMaxPriority=e-Seq(l, x) does not hold.

To prove the only if direction, we prove its contrapositive. Assume we already know that for each x and y has maximal priority in e, if y <pb x, then the rightmost gap-point of x is after call(put, y, p) and call(rm, x). We need to prove that e is MatchedMaxPriority=−e-linearizable. Recall that we already assume that each single-priority execution has FIFO property, and value with larger priority is not covered by values with smaller priority.

Our proof proceed as follows:

- Let e∗ denote the projection of e into operations of priority p. Since each single-priority execution has FIFO property, there exists sequence l∗p, such that e∗ ⊑ l∗p, and when we treat put as enq and rm as deq, l∗p belongs to queue.
- Let a1 be the last inserted value of l∗p.

Step 1: Check whether for each b ∈ Items(e, p), (1) a1 does not <pb to b, and (2) the right-most gap-point of a is after call(put, b, p) and call(rm, b).
It is easy to see that $a_1$ is of priority $p$, and $a_1$ does not $\prec_{pb}$ to any $b \in Items(e, p)$. If for each $b \in Items(e, p)$, the rightmost gap-point of $a_1$ is after $call(put, b, p)$ and $call(rm, b)$. Then by Lemma 33, we can obtain that $e$ is MatchedMaxPriority$^=\cdot$-linearizable.

- Otherwise, there exists $a_2 \in Items(e, p)$, such that the rightmost gap-point of $a_1$ is before $call(put, a_2, p)$ or $call(rm, a_2)$ in $e$. We can see that each gap-point of $a_2$ is after the rightmost gap-point of $a_1$. By assumption, we know that $a_2$ does not $\prec_{pb}$ to $a_1$.

- If for each value $b \in Items(e, p)$, $a_2$ does not $\prec_{pb}$ to $b$. Then we go to step 1 and treat $a_2$ similarly as $a_1$.

- Otherwise, there exists $a_3$ with priority $p$ such that $a_2 <^*_{pb} a_3$.

Since $I_p$ has FIFO property, it is easy to see that there is no cycle in $\prec_{pb}$ order. It is safe to assume that $a_3$ is maximal in the sense of $\prec_{pb}$. Or we can say, there does not exists $a_4$, such that $a_3 <_{pb} a_4$.

By assumption, we know that the rightmost gap-point of $a_3$ is after $call(put, a_2, p)$ and $call(rm, a_2)$. Therefore, we can see that the rightmost gap-point of $a_3$ is after the rightmost gap-point of $a_1$. Then we go to step 1 and treat $a_3$ similarly as $a_1$.

Let $a^i$ be the $a_1$ in the $i$-th loop of our proof. It is not hard to see that, given $i < j$, the rightmost gap-point of $a^i$ is after the rightmost gap-point of $a^j$. Therefore, the loop finally stop at some $a^f$, $a^f$ satisfies the check of Step 1. By Lemma 33, this implies that $e$ is MatchedMaxPriority$^=\cdot$-linearizable. This completes the proof of $if$ direction.

According to the definition of $\prec^*_{pb}$, if $a <^*_{pb} b$, then there exists $a_1, \ldots, a_m$, such that $a <_{pb} a_1 <_{pb} \ldots <_{pb} a_m <_{pb} b$. The following lemma states that, the number of intermediate values $a_i$ is in fact 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 $a, a_1, \ldots, a_m, b$. Then, one of the following holds:

- $a <_{pb} a_1 <_{pb} b$, or $a <^B_{pb} b$;
- $a <_{pb} a_i <_{pb} b$, or $a <^A_{pb} a_i <_{pb} b$, for some $i$.

**Proof.** Our proof proceed as follows:

- ($<^A_{pb}, <^B_{pb}, <^C_{pb}$ and $<^C_{pb}$):

  If $c_3 <_{pb} c_2 <_{pb} c_1$, then $put(c_3, \_)$ happens before $put(c_2, \_)$, and $put(c_2, \_)$ happens before $put(c_1, \_)$.

  Therefore, it is obvious that $put(c_3, \_)$ happens before $put(c_1, \_)$ and $c_3 <_{pb} c_1$.

  Similarly, if $c_3 <_{pb} c_2 <_{pb} c_1$, then $c_3 <_{pb} c_1$.

  If $c_3 <^C_{pb} c_2 <^C_{pb} c_1$. Since $c_2 <^C_{pb} c_1$, $ret(rm, c_2)$ is before $call(put, c_1, \_)$.

  Since $rm(c_2)$ does not happen before $put(c_2, \_)$, $call(put, c_2, \_)$ is before $ret(rm, c_2)$. Since $c_3 <^C_{pb} c_2$, $ret(rm, c_3)$ is before $call(put, c_2, \_)$.

  Therefore, $ret(rm, c_3)$ is before $call(put, c_1, \_)$, and $c_3 <^C_{pb} c_1$.

Therefore, when we meet successive $<^A_{pb}$, it is safe to leave only the first and the last elements and ignore intermediate elements. Similar cases hold for $<^B_{pb}$ and $<^C_{pb}$.

- ($<^A_{pb}$ and $<^C_{pb}$):

  If $c_3 <^A_{pb} c_2 <^C_{pb} c_1$. Since $c_2 <^C_{pb} c_1$, $ret(rm, c_2)$ is before $call(put, c_1, \_)$.

  Since $rm(c_2)$ does not happen before $put(c_2, \_)$, $call(put, c_2, \_)$ is before $ret(rm, c_2)$. Since $c_3 <^A_{pb} c_2$, $ret(put, c_3, \_)$ is before $call(put, c_2, \_)$.

  Therefore, $ret(put, c_3)$ is before $call(put, c_1, \_)$, and $c_3 <^A_{pb} c_1$. 

}
Ahmed Bouajjani, Constantin Enea, and Chao Wang

6

- \( (<A \cdot <B \cdot <A) \): If \( c_4 <_{pb} c_3 <_{pb} c_2 <_{pb} c_1 \). Since \( c_2 <_{pb} c_1 \), \( ret(put, c_2, pand c_1) \) is before \( call(put, c_1, _) \). It is obvious that \( call(put, c_2, _) \) is before \( ret(put, c_2, _) \). Since \( c_3 <_{pb} c_2 \), \( ret(rm, c_3) \) is before \( call(put, c_2, _) \). Therefore, \( ret(rm, c_3) \) is before \( call(put, c_1, _) \), and \( c_3 <_{pb} c_1 \).

\( <_{B} \) and \( <_{C} \):

\( (<B \cdot <C) \): If \( c_3 <_{pb} c_2 <_{pb} c_1 \). Since \( c_2 <_{pb} c_1 \), \( ret(rm, c_2) \) is before \( call(put, c_1, _) \).

Based on above results, given a \( b_1 \), \( b_2 \ldots b_m \) \( a_m <_{pb} b \), where each \( b_i \) is in \( \{A, B, C\} \), we can merge relations, until we get one of the following facts:

- \( a <_{pb} b \) and \( a <_{pb} b \) or \( a <_{pb} b \),
- \( a <_{pb} b \), \( a <_{pb} b \) or \( a <_{pb} b \) for some \( i \).

This completes the proof of this lemma.

There are many enumerations of operations of \( a, b \) and \( a_1 \) that may makes \( a <_{pb} b \). The following lemma states that since some of them is not consistent with the requirements of gap-point, the number of potential enumerations can be further reduced into only five.

\( \textbf{Lemma 34.} \) Given a data-differentated \( p \)-execution \( e \) where Has-MatchedMaxPriority\(^m(e) \) holds. Let \( a \) and \( b \) be values with maximal priority \( p \). Assume that \( a <_{pb} b \), and the rightmost gap-point of \( b \) is before \( call(put, a, p) \) or \( call(rm, a) \). Then, there are five possible enumeration of operations of \( a, b, a_1 \) (if exists), where \( a_1 \) is the possible intermediate value for obtain \( a <_{pb} b \).

\( \textbf{Proof.} \) Let us prove by consider all the possible reasons of \( a <_{pb} b \). According to Lemma [16] we need to consider five reasons: Let \( o \) be the right-most gap-point of \( b \).
Checking Linearizability of Concurrent Priority Queues

- Reason 1, \( a <_{pb} b \):
  Since \( a <_{pb} b \), \( \text{put}(a, p) <_{hb} \text{put}(b, p) \). Since \( o \) is after \( \text{call}(put, b, p) \), and thus, after \( \text{call}(put, a, p) \), we can see that \( o \) is before \( \text{call}(rm, b) \).

  Since single-priority execution must satisfy the FIFO property, \( \text{rm}(b) <_{hb} \text{rm}(a) \), and thus, \( \text{call}(rm, a) \) is before \( \text{ret}(rm, b) \). If \( \text{call}(rm, a) \) is before \( \text{call}(rm, b) \), then \( o \) is also a gap-point of \( a \) and contradicts our assumption. So we know that \( \text{call}(rm, a) \) is after \( \text{call}(rm, b) \). If \( \text{ret}(rm, a) \) is before \( \text{ret}(rm, b) \), since we already assume that there exists gap-point of \( a \), this gap-point is also a gap-point of \( b \), and is after \( o \), which contradicts that \( o \) is the rightmost gap-point of \( b \). Therefore, \( \text{ret}(rm, a) \) is after \( \text{ret}(rm, b) \).

  According to above discussion, there are two possible enumeration of operations of \( a \) and \( b \), as shown in Figure 20 and Figure 21. Here we explicitly draw the leftmost gap-point of \( a \) as \( o' \). Since the position of \( \text{ret}(\text{put}(b, _{)}) \) does not influence the correctness, we can simply ignore it.

- Reason 2, \( a <_{ph} b \):
  Since \( a <_{ph} b \), \( \text{ret}(rm, a) \) is before \( \text{call}(rm, b) \). Since \( o \) is after \( \text{call}(rm, b) \), we can see that \( o \) is before \( \text{call}(put, a, p) \). This implies that \( \text{ret}(rm, a) \) is before \( \text{call}(put, a, p) \), and then \( \text{rm}(a) <_{hb} \text{put}(a) \), which is impossible. Therefore, we can safely ignore this reason.

- Reason 3, \( a <_{ph} b \):
  Since \( a <_{ph} b \), \( \text{ret}(rm, a) \) is before \( \text{call}(put, b, p) \). Since \( o \) is after \( \text{call}(put, b) \), we can see that \( o \) is before \( \text{call}(put, a, p) \). This implies that \( \text{ret}(rm, a) \) is before \( \text{call}(put, a, p) \), and then \( \text{rm}(a) <_{hb} \text{put}(a) \), which is impossible. Therefore, we can safely ignore this reason.

- Reason 4, \( a <_{ph} a_1 <_{ph} b \):
  Since \( a_1 <_{ph} b \), \( \text{rm}(a_1) <_{hb} \text{rm}(b) \), and \( \text{ret}(rm, a_1) \) is before \( \text{call}(rm, b) \). Since \( \text{rm}(a_1) \) does not happen before \( \text{put}(a_1) \), \( \text{call}(put, a_1, p) \) is before \( \text{ret}(rm, a_1) \). Since \( a <_{ph} a_1 \), \( \text{ret}(put, a, p) \) is before \( \text{call}(put, a_1, p) \). Therefore, \( \text{ret}(put, a, p) \) is before \( \text{call}(rm, b) \). Since \( \text{call}(rm, b) \) is before \( o \), we can see that \( o \) is before \( \text{call}(rm, a) \).

  If \( \text{call}(rm, a) \) is after \( \text{ret}(rm, b) \), then \( e \{ a, a_1, b \} \) violates the FIFO property. Therefore, \( \text{call}(rm, a) \) is before \( \text{ret}(rm, b) \). Similarly as the case of reason 1, we can see that \( \text{ret}(rm, b) \) is before \( \text{ret}(rm, a) \).

  According to above discussion, there are three possible enumeration of operations of \( a, a_1 \) and \( b \), as shown in Figure 22, Figure 23 and Figure 24. Here we explicitly draw the leftmost gap-point of \( a \) as \( o' \). Since the position of \( \text{ret}(\text{put}(a_1, _{)}) \) does not influence the correctness, we can simply ignore it. We also ignore \( \text{call}(\text{put}(b, _{)}) \) and \( \text{ret}(\text{put}(b, _{})) \), since the only requirements of them are (1) \( \text{rm}(b) <_{hb} \text{put}(b) \) and (2) \( \text{call}(\text{put}(b, p)) \) is before \( o \).

- Reason 5, \( a <_{ph} a_1 <_{ph} b \):
  Since \( a_1 <_{ph} b \), \( \text{ret}(\text{put}(a_1)) \) is before \( \text{call}(\text{put}(b, p)) \). Since \( \text{call}(\text{put}(b, p)) \) is before \( o \), we can see that \( \text{ret}(\text{put}(a_1, _{})) \) is before \( o \).

- If \( o \) is before \( \text{call}(rm, a) \): Then \( o \) is obviously before \( \text{ret}(rm, a) \). Since \( a <_{ph} a_1, \text{ret}(rm, a) \) is before \( \text{call}(rm, a_1) \). Then we can see that, \( o \) is before \( \text{call}(rm, a_1) \), and remember that \( a_1 <_{ph} b \). Then we can goto the case of reason 1 and treat \( a_1 \) as \( a \). Therefore, we can safely ignore this.

- If \( o \) is before \( \text{call}(\text{put}(a, p)) \): Since \( \text{rm}(a) \) does not happen before \( \text{put}(a, p) \), we can see that \( \text{call}(\text{put}(a, p)) \) is before \( \text{ret}(rm, a) \), and then \( o \) is before \( \text{ret}(rm, a) \). Then similarly as above case, we can see that \( o \) is before \( \text{call}(rm, a_1) \), and \( a_1 <_{ph} b \). Then we can goto the case of reason 1 and treat \( a_1 \) as \( a \). Therefore, we can safely ignore this.

This completes the proof of this lemma.
Figure 20 The first possible enumeration.

Figure 21 The second possible enumeration.

Figure 22 The third possible enumeration.

Figure 23 The forth possible enumeration.

Figure 24 The fifth possible enumeration.
Given a data-differentiated execution $e$ with only one maximal priority, two actions $act_1$, $act_2$ of maximal priority in $e$, and assume that $act_1$ is before $act_2$ in $e$. We say that $act_1$, $act_2$ is covered by values $d_1, \ldots , d_m$ in $e$, if the priorities of $d_1, \ldots, d_m$ is smaller than that of $act_1$ and $act_2$, and

- $\text{ret(put, } d_m, _) \text{ is before } act_1$,
- For each $i < 1 \leq m$, $\text{put}(d_{i-1}, _) <_{pb} \text{rm}(d_i)$,
- $act_2$ is before $\text{call}(\text{rm}, d_1)$.

According to Lemma 15, Lemma 16, and Lemma 34, it is not hard to prove that, given a data-differentiated execution $e$ with only one maximal priority and $\text{has-MatchedMaxPriority}(e)$ holds. $e$ is not MatchedMaxPriority-linearizable, if and only if, one of enumerations holds in $e$ (permit renaming), while $\text{call}(\text{rm}, a)$ and $\text{ret(put, b)}$ is covered by some $d_1, \ldots, d_m$, $\text{call}(\text{rm, b})$ is before $\text{ret(put, d_m, _)}$, and $\text{call}(\text{rm, d_1})$ is before $\text{ret(put, a)}$. We say that such $d_1, \ldots, d_m$ constitute the rightmost gap of $b$.

Let us begin to represent several register automata that is used to capture the existence of a data-differentiated execution $e$, $e$ has a projection $e'$, $e'$ has only one maximal priority, $\text{has-MatchedMaxPriority}(e')$ holds, there exists values $a$ and $b$ with maximal priority in $e'$, $a <_{pb} b$, and the rightmost gap-point of $a$ is before $\text{call(put, a, _)}$ or $\text{call(rm, a)}$.

An automaton $A_{eq}^{\text{1}}$ is given in Figure 25 and it is constructed for the first enumeration in Figure 20. Here we rename the values that cover $\text{call(rm, a)}$ and $\text{ret(put, b)}$ into $d$, and rename the remaining values into $\top$. In this figure, $C = \{\text{call(put, } T, \text{true), ret(put, } T, \text{true), call(rm, } T), \text{call(rm, empty)), ret(rm, empty)}\}$, $C_1 = C \cup \{\text{call(put, } d, < r)\}$, $C_2 = C_1 \cup \{\text{ret(put, } b, = r)\}$, $C_3 = C_2 \cup \{\text{call(put, } d, < r), \text{ret(rm, } d)\}$, $C_4 = C \cup \{\text{ret(put, } b, = r), \text{ret(rm, } d)\}$.

![Figure 25: Automaton $A_{eq}^{\text{1}}$](image)

An automaton $A_{eq}^{\text{2}}$ is given in Figure 26 and it is constructed for the second enumeration in Figure 21. In Figure 26, $C_1, C_2, C_3$ and $C_4$ is same as that in Figure 25.

For the third enumeration in Figure 22. Since we want to ensure that $a$ and $b$ are putted only once, we need to explicitly record the positions of $\text{call(put, } a, p)$ and $\text{call(put, } b, p)$. Since the positions of $\text{call(put, } a, p)$ and $\text{call(put, } b, p)$ are not fixed, there are finite possible cases to consider, as shown below:

- If $\text{call(put, } b, p)$ is after $\text{call(rm, } b)$ and before $\text{call(rm, } a)$: There are two possible positions of $\text{call(put, } a, p)$: (1) before $\text{call(rm, } a_1)$, and (2) after $\text{call(rm, } a_1)$, and before $\text{ret(put, } a, p)$.
- If $\text{call(put, } b, p)$ is after $\text{ret(put, } a_1)$ and before $\text{call(rm, } b)$: same as above case.
Let $A_e$ be the union of register automata that is constructed for above fourteen cases. For example, for the case $c_3$ when $call(put, a, p)$ is before $call(rm, a_1)$, $call(put, b, p)$ is after $ret(put, a_1)$, and $call(put, b, p)$ is before $call(rm, b)$, we construct a finite automaton $A^1_{eq}$ in Figure 27. In Figure 27, let $C$ and $C_1$ the same as that in Figure 25. Let $C_2 = C_1 \cup \{ret(put, a_1, = r)\}$, $C_3 = C_2 \cup \{ret(put, b, = r)\}$, $C_4 = C_3 \cup \{call(put, d, < r), ret(rm, d)\}$, and $C_5 = C \cup \{ret(put, b, = r), ret(put, a_1, = r), ret(rm, d)\}$. Other register automata can be similarly constructed.

Similarly, we construct sets $A^1_{eq}$ and $A^2_{eq}$ of union of register automata for the forth enumeration in Figure 23 and the fifth enumeration in Figure 24 respectively.

Let $A^*_j$ be the union of $A^1_{eq}, A^2_{eq}, A^3_{eq} \cup A^4_{eq}$ and $A^5_{eq}$. The following lemma states that $A^*_j$ is MatchedMaxPriority$^*$-complete.

\textbf{Lemma 35.} $A^*_j$ is MatchedMaxPriority$^*$-complete.

\textbf{Proof.} We need to prove that, given a data-independent implementation $I$, $A^*_j \cap I \neq \emptyset$ if and only if there exists $e \in I$ and $e' \in \text{proj}(e)$ such that $e'$ is not MatchedMaxPriority$^*$-linearizable.

By Lemma 10 and Lemma 15, we need to prove the following fact:

\textbf{Fact 1:} Given a data-independent implementation $I$, $A^*_j \cap I \neq \emptyset$ if and only if there exists $e \in I$ and $e' \in \text{proj}(e)$, Has-MatchedMaxPriority$^*$($e'$) holds, $x$ is the value with maximal priority in $e'$, $e'$ has only one maximal priority, $a$ and $b$ are two values with maximal priority $p$ in $e'$, $a < _p b$ in $e'$, and the rightmost gap-point of $b$ is before $call(put, a, p)$ or $call(rm, a)$ in $e'$.

The only if direction: Assume that $e_1 \in I$ is accepted by some register automata in $A^*_j$. By data-independence, there exists data-differentiated execution $e_2 \in I$ and a renaming function $r$, such that $e_1 = r(e_2)$. Since $e_1$ is accepted by some register automata in $A^*_j$, let $x, y$ and $z$ (if exists) be
the values that are renamed into $b$, $a$, and $a_1$ (if exists) by $r$, respectively, and let $d_1, \ldots, d_m$ be the values that are renamed into $d$ by $r$.

Let $e'' = e_2 \{x, y, z, d_1, \ldots, d_m\}$. It is obvious that $e'' \in \text{proj}(e_2)$, $e''$ has only one maximal priority, and Has-MatchedMaxPriority$^-$ ($e''$) holds. According to our construction of automata in $\text{Auts}_{1\text{-eq}}$, it is not hard to see that $x$ and $y$ have maximal priority in $h_2$, $y <_{pb} x$, and the rightmost gap-point of $x$ is before call(put, $y, p$) or call(rm, $y$) in $e''$.

The if direction: Assume that there exists $e \in \mathcal{I}_\neq$, $e' \in \text{proj}(e)$, such that Has-MatchedMaxPriority$^-$ ($e'$) holds $e'$ has only one maximal priority, $a'$ and $b'$ are two values with maximal priority $p$ in $e'$, $a' <_{pb} b'$ in $e'$, and the rightmost gap-point of $b'$ is before call(put, $a', p$) or call(rm, $a'$) in $e'$. By data-independence, we can obtain execution $e_1$ as follows: (1) rename $a'$ and $b'$ into $a$ and $b$, respectively, (2) for the values $d_1, \ldots, d_m$ that constitute the rightmost gap of $b'$, we rename them into $d$, (3) if $a' <_{pb} a_1' <_{pb} b'$, we rename $a_1'$ into $a_1$, and (4) rename the other values into $\top$. It is easy to see that Has-MatchedMaxPriority$^-$ ($e_1$) holds, $a$ and $b$ have maximal priority in $e_1$, $a <_{pb} b$ in $e_1$, and the rightmost gap-point of $b$ is before call(put, $a, p$) or call(rm, $a$) in $e_1$. By Lemma 34, there are five possible enumeration of operations of $a, b, a_1$ (if exists). Then

- If $a <_{pb} b$ because of the first enumeration, it is easy to see that $e_1$ is accepted by $A^1_{1\text{-eq}}$.
- If $a <_{pb} b$ because of the second enumeration, it is easy to see that $e_1$ is accepted by $A^2_{1\text{-eq}}$.
- If $a <_{pb} b$ because of the third enumeration, it is easy to see that $e_1$ is accepted by some register automaton in $A^3_{1\text{-eq}}$.
- If $a <_{pb} b$ because of the forth enumeration, it is easy to see that $e_1$ is accepted by some register automaton in $A^4_{1\text{-eq}}$.
- If $a <_{pb} b$ because of the fifth enumeration, it is easy to see that $e_1$ is accepted by some register automaton in $A^5_{1\text{-eq}}$.

This completes the proof of this lemma.

\[\square\]
D.5 UnmatchedMaxPriority\(>\) (e) Always Holds

Lemma 36. Given a data-differentiated execution e with only one maximal priority, if Has-UnmatchedMaxPriority\(>\) (e) holds, then e \(\sqsubseteq l\) for some l where UnmatchedMaxPriority\(>\)-Seq(l, x) holds.

Proof. Since Has-UnmatchedMaxPriority\(>\) (e) holds, the actions with maximal priority in e is some unmatched put. Therefore, no matter how we locate linearization points, we can always obtain a sequence l of operations that contains unmatched put with maximal priority, and this satisfy the requirements of UnmatchedMaxPriority\(>\). This completes the proof of this lemma.

D.6 UnmatchedMaxPriority\(=\) (e) Always Holds

Lemma 37. Given a data-differentiated execution e with only one priority and Has-UnmatchedMaxPriority\(=\) (e) holds. e is not UnmatchedMaxPriority\(=\) (e)-linearizable, if and only if there exists x and y with maximal priority p, x has unmatched put, y has matched put and rm, and put(x, p) \(\prec_{hb}\) put(y, p).

Proof. The if direction is obvious.

To prove the only if direction, we prove its contrapositive. Assume that for each pair of x and y with maximal priority in e, if x has unmatched put, y has matched put and rm, then put(x, p) does not happen before put(y, p). We need to prove that e is UnmatchedMaxPriority\(=\) (e)-linearizable.

Let \(x_1, \ldots, x_m\) be the set of values with priority p and has unmatched put in e, let \(y_1, \ldots, y_n\) be the set of values with priority p and has matched put and rm in e. By assumption, we know that \(\text{call}(put, y_i, p)\) is before \(\text{ret}(put, x_j, p)\) for each i, j. Then we explicitly construction the linearization of e by locating the linearization points of e as follows:

- For each \(x_i\), locate the linearization point of \(put(x_i, p)\) just before its return action.
- For each \(y_j\), locate the linearization point of \(put(y_j, p)\) just after its call action.
- For other operations, locate their linearization points at an arbitrary location after its call action and before its return action.

Let l be the sequence of linearization points. It is easy to see that e \(\sqsubseteq l\). Since linearization points of \(put(x_i, p)\) is after the linearization point of \(put(y_j, p)\) for each i, j, it is easy to see that UnmatchedMaxPriority\(=\)-Seq(l, x') holds where x' = \(x_m\) and its return action is after other values in \(x_1, \ldots, x_m\). This completes the proof of this lemma.

Lemma [37] shows how to check non UnmatchedMaxPriority\(=\) (e)-linearizable exections. However, the case in Lemma [37] violates our assumption that each single-priority execution is FIFO. Therefore, we know that UnmatchedMaxPriority\(=\) (e) always holds, as states by the following lemma.

Lemma 38. Given a data-differentiated execution e with only one maximal priority, if Has-UnmatchedMaxPriority\(=\) (e) holds, then e \(\sqsubseteq l\) for some l where UnmatchedMaxPriority\(=\)-Seq(l, x) holds.

Proof. According to Lemma [37] if Has-UnmatchedMaxPriority\(=\) (e) holds and e is not UnmatchedMaxPriority\(=\)-linearizable, then there exists x and y with maximal priority p, x has unmatched put, y has matched put, and put(x, p) \(\prec_{hb}\) put(y, p). Let \(e_1 = e \{ (x, y) \}\). It is obvious that \(e_1\) does not satisfy FIFO property. This contradicts the assumption that every single-priority execution has FIFO property, and thus, we can safely ignore this case.
D.7 Proofs, Definitions and Register Automata for EmptyRemove

In this subsection we construct EmptyRemove-complete register automata. The notion of left-right constraint of rm(empty) is inspired by left-right constraint of queue \[5\].

**Definition 39.** Given a data-differentiated execution \(e\), and \(o = rm(\text{empty})\) of \(e\). The left-right constraint of \(o\) is the graph \(G\) where:

- the nodes are the values of \(e\) or \(o\), to which we add a node,
- there is an edge from value \(d_1\) to \(o\), if \(\text{put}(d_1, \_ )\) happens before \(o\),
- there is an edge from \(o\) to value \(d_1\), if \(o\) happens before \(rm(d_1)\) or \(rm(d_1)\) does not exists in \(h\),
- there is an edge from value \(d_1\) to value \(d_2\), if \(\text{put}(d_1, \_ )\) happens before \(rm(d_2)\).

Given a data-differentiated execution \(e\) and \(o = rm(\text{empty})\) of \(e\), it is obvious that Has-EmptyRemove\(^-\)(\(e\)) holds. Let \(USet_e(e, o) = \{ \text{op} \mid \text{op} \text{ is an operation of some value, and either } \text{op} \leq_{hb} o \text{, or there is } \text{op}' \text{ with the same value of } \text{op}, \text{such that } \text{op}' <_{hb} o \}. \) For each \(i \geq 1\), let \(USet_{i+1}(e, o) = \{ \text{op} \mid \text{op} \text{ is an operation of some value, } \text{op} \text{ is not in } USet_i(e, o) \text{ for each } k \leq i \text{, and either } \text{op} \text{ happens before some } \text{op}' \in USet_i(e, o) \text{, or there is } \text{op}'' \text{ with the same value of } \text{op} \text{ and } \text{op}'' \text{ happens before some } \text{op}' \in USet_i(e, o) \}. \) We can see that \(USet_i(e, o) \cap USet_{i+1}(e, o) = \emptyset\) for any \(i \neq j\). Let \(USet(e, o) = USet_1(e, o) \cup USet_2(e, o) \cup \ldots\)

Similarly as UVSet, we can prove the following two lemmas for USet.

**Lemma 40.** Given a data-differentiated execution \(e\) where Has-EmptyRemove\(^-\)(\(e\)) holds. Let \(o\) be a rm(\(\text{empty}\)) of \(e\). Let \(G\) be the graph representing the left-right constraint of \(o\). Assume that \(G\) has no cycle going through \(o\). Then, USet\((e, o)\) contains only matched put and rm.

This Lemma can be similarly proved as Lemma\([30]\).

**Lemma 41.** Given a data-differentiated execution \(e\) where Has-EmptyRemove\(^-\)(\(e\)) holds. Let \(o\) be a rm(\(\text{empty}\)) of \(e\). Let \(G\) be the graph representing the left-right constraint of \(o\). Assume that \(G\) has no cycle going through \(o\). Then, \(o\) does not happen before any operation in USet\((e, o)\).

This Lemma can be similarly proved as Lemma\([31]\).

Then we can prove that getting rid of cycle though \(o\) in left-right constraint is enough for ensure linearizable w.r.t MS(SeqPQ), as stated by the following lemma.

**Lemma 42.** Given a data-differentiated execution \(e\) where Has-EmptyRemove\(^-\)(\(e\)) holds. \(e\) is not EmptyRemove-linearizable, if and only if there exists \(o = rm(\text{empty})\) in \(e\), \(G\) has a cycle going through \(o\), where \(G\) is the graph representing the left-right constraint of \(o\).

**Proof.** To prove the if direction, assume that there is such a cycle. Assume by contradiction that \(e \subset l = u \cdot v\) and EmptyRemove-Seq\((l, o)\) holds. Let \(U\) and \(V\) be the set of operations in \(u\) and \(v\). Let the cycle be \(d_1 \rightarrow d_2 \rightarrow \ldots \rightarrow d_m \rightarrow o \rightarrow d_1\) in \(G\). Since \(d_m \rightarrow o\), put\((d_m, \_ )\) happens before \(o\), and it is easy to see that put\((d_m, \_ )\) is in \(U\). Since \(U\) contains matched put and rm, we can see that operations of \(d_m\) is in \(U\). Similarly, we can see that operations of \(d_{m-1}, \ldots, d_1\) is in \(U\). If \(rm(d_1)\) does not exists, then this contradicts that \(U\) contains matched put and rm. Else, if \(rm(d_1)\) exists, since \(o\) happens before \(rm(d_1)\), we can see that \(rm(d_1) \in V\), which contradicts that \(rm(d_1) \in U\).

This completes the proof of the if direction.

To prove the only if direction, we prove its contrapositive. Assume that for each such \(o\) and \(G\), \(G\) has no cycle going through \(o\). Let \(O\) be the set of operations of \(e\), except for \(rm(\text{empty})\). Let \(O_L = USet(e, o) \cup O\), \(O_R = O \setminus O_L\).

By Lemma\([40]\) we can see that \(O_L = USet(e, o)\) contains only matched put and rm. Let \(O'_L\) be the union of \(O_L\) and all the \(rm(\text{empty})\) that happens before some operations in \(O_L \cup \{o\}\). Let \(O'_R\)
be the union of $O_R$ and the remaining $rm(empty)$. It remains to prove that for $O'_L$, $\{o\}$, $O'_R$, no elements of the latter set happens before elements of the former set. We prove this by showing that all the following cases are impossible:

- Case 1: If some operation $o_r \in O'_R$ happens before $o$. Then we can see that $o_r \in USet(e,o)$ or is a $rm(empty)$ that happens before $o$, and then $o_r \in O'_R$, which contradicts that $o_r \in O'_R$.
- Case 2: If some operation $o_r \in O'_R$ happens before some operation $o_l \in O'_L$. Then we know that $o_r \in USet(e,o)$ or is a $rm(empty)$ that happens before some operations in $O_L \cup \{o\}$, and then $o_r \in O'_L$, which contradicts that $o_r \in O'_R$.
- Case 3: If $o$ happens before some $o_l \in O'_L$. If $o_l \in USet(e,o)$, then by Lemma 41 we know that this is impossible. Else, $o_l$ is a $rm(empty)$ that happens before some operations in $O_L \cup \{o\}$, and $o$ happens before some operations in $O_L \cup \{o\}$, which is impossible by Lemma 41.

This completes the proof of the only if direction.

Let us begin to represent a register automaton that is used for capture the case that, in a sub-execution $e'$ of an execution $e$, Has-EmptyRemove($e'$) holds, $e'$ is not EmptyRemove-linearizable, and the reason is that there is a cycle going through some $rm(empty)$ $o$ in the left-right constraint of $o$. The automaton is $A_{SeqPQ}^3$, which is given in Figure 28. In Figure 28 let $C = \{\text{call}(put, \top, \text{true}), \text{ret}(put, \top, \text{true}), \text{call}(rm, \top), \text{ret}(rm, \top), \text{call}(rm, empty), \text{ret}(rm, empty)\}$, $C_1 = C \cup \{\text{call}(put, b, true)\}$, $C_2 = C_1 \cup \{\text{ret}(rm, b)\}$, and $C_3 = C \cup \{\text{ret}(rm, b)\}$.

**Figure 28 Automaton $A_{SeqPQ}^3$**

Given a data-differentiated execution $e$, we say that $o = rm(empty)$ in $e$ is covered by values $d_1, \ldots, d_m$, if

- put($d_m, _$) happens before $o$,
- For each $i < m$, put($d_i, _$) happens before $rm(d_i)$,
- $o$ happens before $rm(d_i)$, or $rm(d_i)$ does not exists in $e$

According to the definition of left-right constraint for $o$, in a data-differentiated execution $e$, there is a cycle going through $o$, if and only if there exists values $d_1, \ldots, d_m$, such that $o$ is covered by $d_1, \ldots, d_m$.

**Lemma 43.** $A_{SeqPQ}^3$ is EmptyRemove$^=\$-complete.

**Proof.** We need to prove that, given a data-independent implementation $\mathcal{I}$. $A_{SeqPQ}^3 \cap \mathcal{I} \neq \emptyset$ if and only if there exists $e \in \mathcal{I}$ and $e' \in \text{proj}(e)$ such that $e'$ is not EmptyRemove$^=\$-linearizable.

By Lemma 42, we need to prove the following fact:
Therefore, we introduce a specific value put succeeds until some item is removed. However, we assume that each method can always make progress in isolation, which means that put must return some value when the priority queue is full. Then, at some time, the priority queue may become full and then put cannot be obtained from \( e \). By renaming \( d_1, \ldots, d_m \) into \( b \) and renaming other values into \( \top \). By data-independence, \( e_1 \in I \).

It is easy to see that \( e_1 \) is accepted by \( A^3_{\text{SeqPQ}} \).

This completes the proof of this lemma.

### D.8 Proofs and Definitions in Section 4.2

In this subsection, we show how to prove that the verification of linearizability of priority queue to be PSPACE-complete for a fixed number of threads, and EXPSPACE-complete for unbounded number of processes.

When considering decidability and complexity, we consider only finite variables with finite data domain. In such case, the priority queue implementations can only model priority queue with bounded capacity. Then, at some time, the priority queue may become full and then put can not succeed until some item is removed. However, we assume that each method can always make progress in isolation, which means that put must return some value when the priority queue is full.

Therefore, we introduce a specific value full. A put method returns full indicates that the priority queue is full now. To fit our results of priority queue in former section, when checking linearizability of priority queue, we only consider executions without any put(\_,\_,\_\_,\_\_,full). Since does not mean does not mean we consider only executions with bounded number of operations, since an execution can contain unbounded number of pairs of put and remove operations by linearizable w.r.t pairs of put and remove operations.

**Vector Addition Systems with States (VASS):** A VASS \([4]\) \( A = (Q_A, \rightarrow_A) \) contains a finite set \( Q_A \) of states and a finite set \( \rightarrow_A \) of transitions. Each transition is chosen from \( Q_A \times \mathbb{Z}^s \times Q_A \), where \( s \) is a positive integer. A vector that has a single non-zero component in \( \{1, -1\} \) is called a unit vector. We assume, w.l.o.g., that all vectors of transitions are unit vectors.

The operational semantics of a VASS is defined as a LTS. A configuration \((q, \vec{n})\) of VASS \( A \) contains a state \( q \in Q_A \) with a vector \( \vec{n} \in \mathbb{N}^s \). \( \vec{n} \) can be considered as values of several counters. We use \( \vec{n}(i) \) to denote the \( i \)-th component of \( \vec{n} \). The transitions between configurations of VASS is defined as follows: \((q_1, \vec{n}_1) \rightarrow_A (q_2, \vec{n}_2)\), if there exists vector \( \vec{n}_1 \), such that (1) \((q_1, \vec{n}, q_2) \in \rightarrow_A\), (2) if \( \vec{n}(j) = 1 \) for some \( j \), then \( \vec{n}_2 \) is obtain from \( \vec{n}_1 \) by increasing the \( i \)-th component by 1, and (3) if \( \vec{n}(j) = -1 \) for some \( j \), then \( \vec{n}_2 \) is obtain from \( \vec{n}_1 \) by decreasing the \( i \)-th component by 1.

The state-reachability problem of VASS is to determine whether some configuration with state \( q \) is reachable from a given configuration \((q_0, 0)\). Here \( 0 \) is a specific vector for which all components are 0 and \( q_0 \) is the initial state of VASS.

**Reducing priority Queue into PSPACE and EXPSPACE problems:** We describe a class \( C \) of data-independent implementations for which linearizability w.r.t. SeqPQ is decidable. The implementations in \( C \) 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 \( D \). Methods communicate through a finite number of shared variables that also store Boolean values, or data values from \( D \). 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 $\Gamma$-complete automata $A(\Gamma)$ uses a fixed set $D = \{a, b, a_1, d, e, \top\}$ of values, we have that $C \cap A(\Gamma) \neq \emptyset$ for some $\Gamma$ if $C_D \cap A(\Gamma) \neq \emptyset$ where $C_D$ is the subset of $C$ that uses only values in $D$.

We model the set $C_D$ as the executions of a Vector Addition System with States (VASS) similarly to Bouajjani et al. [4]. A VASS has a finite set of states, and manipulates a finite set of counters holding non-negative values. The idea is that the states of the VASS represent the global variables of $C_D$. 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.

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. In this way, we reduce checking linearizability of priority queue into the the EXPSPACE-complete problem of checking state-reachability in a VASS. When we consider only finite number of threads, the value of counters are bounded and then this problem is PSPACE [10].

Reducing PSPACE and EXPSPACE problems into linearizability of priority queue for bounded number of threads: We show how to reduce the state-reachability problem of VASS into checking linearizability of priority queue. Our construction is based on the reduction of state-reachability problem of counter machine and VASS into linearizability in [4].

There are already several linearizable lock-free implementations of priority queue [21, 17]. It is not hard to modify them to fit finite variables with finite data domain and it is obvious that they are still linearizable then. Let Imp-PQ$_f$ be one such implementations. It is safe to assume that Imp-PQ$_f$ is given in the form of pseudo-code. We use put$_q$ and rmn$_q$ to explicitly denote the pseudo-code of put and remove method of Imp-PQ$_f$.

A counter machine is a VASS that can additional detect if the value of some counter is 0. In [4], they construct a library $L_A$ that simulate executions of counter machine $A$. This library use finite variables with finite data domain. It contains the following six methods:

- $M(q, \bar{n}, q')$, which represents a transition resulting of $(q, \bar{n}, q')$.
- $M(q, i, q')$, which represents a transition that from state $q$ to state $q'$ while detecting the value of counter $i$ to be 0.
- $M_{qf}$, This method returns if the simulating reaches some state with state $q_f$.
- $M_{inc_i}$, which simulates increasing counter $i$ with 1.
- $M_{dec_i}$, which simulates decreasing counter $i$ with 1.
- $M_{zmi}$, which simulates testing whether the value of counter $i$ to be 0.

The first three methods do their work with the help of the latter three methods. For example, a $M(q, \bar{n}, q')$ method (that increases counter $j$) will activate a $M_{inc-j}$ method and a $M_{dec-j}$ method.

Let $D' = \{(q, \bar{n}, q')|q, q' \in Q_A \setminus \{q_f\}, \bar{n} \in \mathbb{Z}^s \} \cup \{q_f\} \cup \{inc-i, dec-i|1 \leq i \leq s\}$. Let RAs be the register automata obtained as follows:

- Given $\Gamma \in \{EmptyRemove, UnmatchedMaxPriority, MatchedMaxPriority\}$, we modify the $\Gamma$-complete automata while the original value of $\top$ is replaced by values $\top$ and all values in $D'$.
- For register automata that checks FIFO of sub-executions of operations with a same priority, if the original value under consideration is $\{b\}$ (the first three automata) or $\{a, b\}$ (the forth automaton), then we replace them with $\{b\} \cup D'$ and $\{a, b\} \cup D'$, respectively.
Given a VASS $A$, we construct a priority queue implementation $PQ_A$ as follows:

- A new memory location $reachQf$ is introduced. Its initial value is set to false.
- The $rm$ method first check the value of $reachQf$. If it is true, then it returns a random value. Otherwise, it works as $rm_f$. The pseudo-code of $rm$ is shown in Algorithm 2.

- The $put$ method uses items from $D' \cup \{a, b, a_1, d, e, \top\}$. It first uses $put_f$ to do the work of priority queue, and then choose a role and begin to simulate transitions of $A$, until $q_f$ is reached. Then, it set the flag $reachQf$ to be true. The pseudo-code of $put$ is shown in Algorithm 3.

In [4], to use a library to simulate counter machine $A$, their library $L_A$ contains the following variables: (1) variable $q$ that uses value from $Q_A$, (2) vector $req[]$ and $ack[]$, the index is chosen from 1 to the number of unit vectors of length $s$, (3) $dec[]$, where the index is chosen form 1 to $s$.

Let us explain how $put$ method do to act according to its chosen role. We use the case of $\text{trans}(q, \vec{n}, q')$ as an example while other cases can be similarly obtained. For the role $\text{trans}(q, \vec{n}, q')$, it works as $M(q, \vec{n}, q')$ in $L_A$. Or we can say, it do the following work:

- atomically do $\text{wait}(q)$ and $\text{signal}(req[\vec{n}])$,
- atomically do $\text{wait}(ack[\vec{n}])$ and $\text{signal}(q')$,

Note that we want to obtain a implementation where each method can always make progress in isolation. To obtain such an implementation, we need to slightly modify the $wait$ command as follows:

- If a wait command find itself blocked, then it can still proceed. However, in this case the $put$ method will return a specific value $fail$. Moreover, when deciding linearizability, we do not consider executions with return value $fail$.

Follows the proof in [4], we can prove that $(q_f, \_)$ is reachable in $A$, if and only if $PQ_A$ is not linearizable w.r.t SeqPQ. In this way, we reduce the state-reachability problem of VASS into checking linearizability of priority queue.

With above discussions, we can now prove the following theorem:

**Theorem 17** Verifying whether an implementation in $C$ is linearizable w.r.t. SeqPQ is PSPACE-complete for a fixed number of threads, and EXPSPACE-complete otherwise.
Algorithm 3: put of $PQ_A$

**Input:** An item $itm$ and its priority $p$

**Output:** $full$ if the priority queue is full

1. $x = put_f(itm, p)$;
2. random select a role from \{trans($q, \bar{n}, q'$), reach-$q_f$, inc-$i$, dec-$i$\};
3. works according to chosen role;
4. if $q_f$ is reached then
5.   $reachQf := true$;
6. if $x = full$ then
7.   return $x$;
8. else
9.   return;