Checking Linearizability of Concurrent Priority Queues

Ahmed Bouajjani¹, Constantin Enea², and Chao Wang³

1 IRIF, University Paris Diderot, Paris, France
   abou@irif.fr
2 IRIF, University Paris Diderot, Paris, France
   cenea@irif.fr
3 IRIF, University Paris Diderot, Paris, France
   wangch@irif.fr

Abstract

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

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Concurrency, Linearizability, Model Checking

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.16

1 Introduction

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 machine instructions like compare-and-swap. Intuitively, linearizability asks that every individual operation appears to take place instantaneously at some point between its invocation and its return. Verifying linearizability is intrinsically hard, and undecidable in general [4]. However, recent work [5] has shown that for particular objects, e.g., registers, mutexes, queues, and stacks, the problem of verifying linearizability becomes decidable (for finite-state implementations).

In this paper, we consider another important object, namely the priority queue, which is essential for applications such as task scheduling and discrete event simulation. Numerous implementations have been proposed in the research literature, e.g., [2, 8, 16, 20, 19], and concrete implementations exist in many modern languages like C++ or Java. Priority queues are collections providing put and rm methods for adding and removing values. Every added value is associated to a priority and a remove operation returns a minimal priority value.

* This work is supported in part by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 678177).
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 queues, values are removed in the order in which they were inserted), but by parameters of the put operations (the priorities) which come from an unbounded domain. For instance, the sequential behavior $\text{put}(a,p_1) \cdot \text{put}(b,p_3) \cdot \text{put}(c,p_2) \cdot \text{rm}(a,p_1) \cdot \text{rm}(c,p_2)$ 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.

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

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

Next, we show that checking whether an execution violates this “local” property for a value $x$ can be done using a class of register automata [15, 9, 18] (transition systems where the states consist of a fixed set of registers that can receive values and be compared). Actually, only two registers are needed: one register $r_1$ for storing a priority guessed at the initial state, and one register $r_2$ for reading priorities as they occur in the execution and comparing them with the one stored in $r_1$. We show that registers storing values added to or removed from the priority queue are not needed, since any data-independent implementation admits a violation to linearizability whenever it admits a violation where the number of values is constant, and at most 4 (the number of priorities can still be unbounded).

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

We consider priority queues whose interface contains two methods put and rm for adding and respectively, removing a value. Each value is assigned with a priority when being added to the data structure (by calling put) and the remove method rm removes a value with a minimal priority. For generality, we assume that the set of priorities is partially-ordered. Incomparable priorities can be removed in any order. When multiple values are assigned with the same priority, rm returns the least recent value. Also, when the set of values stored in the priority queue is empty, rm returns the distinguished value empty. In this section, we formalize (concurrent) executions and implementations, introduce a set of properties satisfied by all the implementations we are aware of, and recall the standard correctness criterion for concurrent implementations of ADTs known as linearizability [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 \( \text{call}_o(\text{put}, a, p) \) and \( \text{call}_o(\text{rm}, a') \) 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 \(^1\). The return actions are denoted in a similar way as \( \text{ret}_o(\text{put}, a, p) \) and respectively, \( \text{ret}_o(\text{rm}, a') \).

An execution \( e \) is a sequence of call and return actions which satisfy the following well-formedness properties: each return is preceded by a matching call (having the same operation identifier), and each operation identifier is used in at most one call/return. We assume every set of executions is closed under isomorphic renaming of operation identifiers. An \( m(a) \)-operation in an execution \( e \) is an operation identifier \( o \) s.t. \( e \) contains the actions \( \text{call}_o(m, a) \) and \( \text{ret}_o(m, a) \). An execution is called sequential when no two operations overlap, i.e., each call action is immediately followed by its matching return action, and concurrent otherwise.

For readability, we write a sequential execution as a sequence of \( \text{put}(a, p) \) and \( \text{rm}(a) \) symbols representing a pair of actions \( \text{call}_o(\text{put}, a, p) \cdot \text{ret}_o(\text{put}, a, p) \) and \( \text{call}_o(\text{rm}, a) \cdot \text{ret}_o(\text{rm}, a) \), respectively (\( o \in O \)). For example, given two priorities \( p_1 \prec p_2 \), \( \text{put}(a, p_2) \cdot \text{put}(b, p_1) \cdot \text{rm}(b) \) is a sequential execution of the priority queue (\( \text{rm} \) returns \( b \) because it has smaller priority).

We define \( \text{SeqPQ} \), the set of sequential priority queue executions, semantically via a labelled transition system (LTS, for short). An LTS is a tuple \( A = (Q, \Sigma, \rightarrow, q_0) \), 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 the priority queue as an LTS \( PQ \) where states are mappings 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), and the transition labels are \( \text{put}(a, p) \) and \( \text{rm}(a) \). 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 \( e \). Then, \( \text{SeqPQ} \) is the set of traces (words) accepted by \( PQ \).

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

---

\(^1\) Method return values are guessed nondeterministically, and validated at return points. This can be handled using assume statements, which only admit executions satisfying a given predicate.
completed executions, where each call action has a corresponding return action. This simplification is sound when the method invocations can always make progress in isolation.

### 2.2 Semantic Properties of Priority Queues

We define two properties which are important for our results: (1) **data independence** [22, 1] states that priority queue behaviors do not depend on the actual values which are added to the queue, and (2) **closure under projection** [5] states that executions remain valid by removing all the operations adding or removing certain values.

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

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 keeps the priorities unchanged. Intuitively, renaming values has no influence on the behavior of the priority queue, contrary to renaming priorities.

**Definition 1.** A set of executions $E$ is **data independent** iff

- for all $e \in E$, there exists $e' \in E_{\neq}$ and a renaming function $r$, such that $e = r(e')$,
- for all $e \in E$ and for all renamings $r$, $r(e) \in E$.

The following lemma is a direct consequence of definitions.

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

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

Besides data independence, the sequential executions of the priority queue satisfy the following closure property: an execution remains valid when removing all the operations with an argument in some set of values $D \subseteq D$ and any $\text{rm}(\text{empty})$ operation (since they are read-only and they don’t affect the queue’s state). 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 $\text{call}_o(\text{rm}, \text{empty}, a)$ 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 the call/return actions with an argument not in $D$. We write $e \setminus x$ for the projection $e|_{D \setminus \{x\}}$.

**Lemma 3.** 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 happen-before relation $<_{hb}$ between operations $^2$ is defined as follows: $o_1 <_{hb} o_2$, if the return action of $o_1$ occurs before...
the call action of \( o_2 \) in \( e \). The happens-before relation is an interval order [6]: for distinct \( o_1, o_2, o_3, o_4 \), if \( o_1 <_{hb} o_2 \) and \( o_3 <_{hb} o_4 \), then either \( o_1 <_{hb} o_4 \), or \( o_3 <_{hb} o_2 \). Intuitively, this comes from the fact that concurrent threads share a notion of global time.

Given a (concurrent) execution \( e \) and a sequential execution \( s \), we say that \( e \) is linearizable w.r.t \( s \), denoted \( e \sqsubseteq s \), if there is a bijection \( f : O_1 \rightarrow O_2 \), where \( O_1 \) and \( O_2 \) are the set of operations of \( e \) and \( s \), respectively, such that (1) the call and return actions with identifier \( o \) and \( f(o) \), respectively, are the same and (2) if \( o_1 <_{hb} o_2 \), then \( f(o_1) <_{hb} f(o_2) \). A (concurrent) execution \( e \) is linearizable w.r.t. a set \( S \) of sequential executions, denoted \( e \sqsubseteq S \), if there exists \( s \in S \) such that \( e \sqsubseteq s \). A set of concurrent executions \( E \) is linearizable w.r.t. \( S \), denoted \( E \sqsubseteq S \), if \( e \sqsubseteq S \) for all \( e \in E \).

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

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

## 3 Checking Linearizability of Priority Queue Executions

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

### 3.1 Characterizing Data-Differentiated Sequential Executions

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

We explain how the procedure works on the following execution:

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

(1)

where \( p_1, p_2, p_3 \) are priorities such that \( p_1 < p_2 \) and \( p_1 < p_3 \), and \( p_2 \) and \( p_3 \) are incomparable. Since the \( \text{rm}(\text{empty}) \) operations are read-only (they don’t affect the queue’s state), they are selected first. An \( \text{rm}(\text{empty}) \)-operation \( o \) is correct when 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 \( x \in \{a, c\} \). Thus, the correctness of (1) reduces to the correctness of

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

(2)

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

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

(3)
Checking Linearizability of Concurrent Priority Queues

Algorithm 1: Check-PQ-Seq

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

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

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

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

Has-EmptyRemoves($e$) = true iff $e$ contains a rm(empty)-operation

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

where priorities($e$), resp., unmatched-priorities($e$), is the set of priorities occurring in put operations of $e$, resp., in put operations of $e$ for which there is no rm operation removing the same value. We call the latter unmatched put operations. A put operation which is not unmatched is called matched. For simplicity, we consider the following syntactic sugar

Has-MatchedMaxPriority($e$) = $\neg$Has-EmptyRemoves($e$) $\land \neg$Has-UnmatchedMaxPriority($e$).

By an abuse of notation, we assume Has-UnmatchedMaxPriority($e$) $\Rightarrow \neg$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:

EmptyRemove-Seq($e, o$) = true iff $e = u \cdot o \cdot v$ and matched($u$)

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

$\quad p \not\in$ priorities($v$)

MatchedMaxPriority-Seq($e, x$) = true iff $e = u \cdot \text{put}(x, p) \cdot v \cdot \text{rm}(x) \cdot w$, matched$_\prec$($u \cdot v \cdot w$),

$\quad p \not\in$ unmatched-priorities($u \cdot v \cdot w$), $p \not\in$ priorities($u \cdot v \cdot w$), and $p \not\in$ priorities($v \cdot w$)

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

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

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

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

### 3.2 Checking Linearizability of Data-Differentiated Concurrent Executions

The extension of Check-PQ-Seq to concurrent executions, checking whether they are linearizable w.r.t. \( \text{SeqPQ} \), is obtained by replacing every predicate \( \Gamma\text{-Seq} \) with

\[
\Gamma\text{-Conc}(e, α) = \text{true} \text{ iff there is a sequential execution } s \text{ such that } e \subseteq s \text{ and } \Gamma\text{-Seq}(s, α)
\]

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

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

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

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

We explain the proof of this property on the execution \( e \) in Figure 1(a) where \( p_1 < p, p_1 < p_2 \), and the predicate Has-MatchedMaxPriority(e) holds. Assume that there exist two sequential executions \( l \) and \( l' \) such that \( e \subseteq l = u \cdot \text{put}(x, p) \cdot v \cdot \text{rm}(x) \cdot w \), MatchedMaxPriority-Seq(l, x) holds, and \( e \setminus x \subseteq l' \subseteq \text{SeqPQ} \). Let \( u = e, w \) be any sequence formed of \( \text{put}(z_2, p_2) \) and \( \text{rm}(z_1) \) (we distinguish them by adding the suffix “−w” to their name, e.g., \( \text{rm}(z_1) - w \)), and \( v \) be any sequence containing the remaining operations. In general, the linearization \( l' \) can be defined by choosing for each operation, a point in time between its call and return, called **linearization point**. The order between the linearization points defines the sequence \( l' \). Figure 1(a) draws linearization points for the operations in \( e \setminus x \) which define \( l' \).

An operation is called \( p \)-comparable (resp., \( p \)-incomparable) when it receives as argument a value of priority comparable to \( p \) (resp., incomparable to \( p \)). Defining \( l''_1, l''_2, l''_3 \) as the projection of \( l' \) to the set of operations in \( u, v \) and \( w \), respectively, leads to a sequence \( l'' \not\subseteq \text{SeqPQ} \). This is because MatchedMaxPriority-Seq(l, x) imposes no restriction on \( p \)-incomparable operations in \( u \cdot v \), and the projection of \( l' \) to \( p \)-incomparable operations in \( u \cdot v \) is not in \( \text{SeqPQ} \). In this example, this projection is \( \text{put}(z_2, p_2) \cdot \text{rm}(z_2) \).

---

3 In general, there may exist multiple ways of choosing linearization points to define the same linearization. Our construction is agnostic to this choice.
We define the sets of operations $U'$, $V'$ and $W'$ such that $l'_1$, $l'_2$ and $l'_3$ are the projections of $l$ to $U'$, $V'$, and $W'$, respectively. This is done in two steps:

1. 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 $l'$.

2. 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 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 $l''_1 = \text{put}(z_1, p_2) \cdot \text{put}(x_2, p)$, $l''_2 = \text{put}(z_2, p_2) \cdot \text{rm}(x_2) \cdot \text{put}(y_1, p_1) \cdot \text{rm}(y_1)$, and $l''_3 = \text{rm}(z_1) \cdot \text{rm}(z_2)$. Figure 1(b) draws linearization points for each operation in $e$ defining the linearization $l''$. Figure 1(b) exemplifying the soundness of $\text{Check-PQ-Conc}$.

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

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

4 Reducing Linearizability of Priority Queues to Reachability

We show that the set of executions for which $\text{Check-PQ-Conc-NonRec}$ fails on some projection can be described using register automata, modulo a value renaming. Renaming values (which is complete under data independence) allows to simplify the reasoning about projections. W.l.o.g., we assume that all the operations which are not in the projection failing this test use the same distinguished value $\top$, different from those in the projection. Then, it is enough to find an automata characterization of the executions $e$ for which
Check-PQ-Conc-NonRec(e) is false, i.e., for which \( \Gamma(e) := \text{Has}-\Gamma(e) \Rightarrow \exists \alpha. \Gamma\text{-Conc}(e, \alpha) \) is false, for some \( \Gamma \in \{\text{EmptyRemove}, \text{UnmatchedMaxPriority}, \text{MatchedMaxPriority}\} \). Intuitively, \( \Gamma(e) \) states that \( e \) is linearizable w.r.t. the set of sequential executions described by \( \Gamma\text{-Seq} \) (provided that \( \text{Has}-\Gamma(e) \) holds). Therefore, by an abuse of terminology, an execution \( e \) satisfying \( \Gamma(e) \) is called linearizable w.r.t. \( \Gamma \), or \( \Gamma\text{-linearizable} \). Extending the automaton characterizing non \( \Gamma\text{-linearizable} \) executions with self-loops that allow any operation with argument \( \Gamma \) results in an automaton satisfying the following property called \( \Gamma\text{-completeness} \).

\[ \Gamma\text{-complete} \]

For \( \Gamma \in \{\text{EmptyRemove}, \text{UnmatchedMaxPriority}, \text{MatchedMaxPriority}\} \), an automaton \( A \) is called \( \Gamma\text{-complete} \) when for each data-independent implementation \( I \):

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

We can show that for any \( \Gamma \in \{\text{EmptyRemove}, \text{UnmatchedMaxPriority}, \text{MatchedMaxPriority}\} \), there exists a \( \Gamma\text{-complete} \) automaton. For lack of space, we only consider the case \( \Gamma = \text{MatchedMaxPriority} \) in Section 4.1. When defining \( \Gamma\text{-complete} \) 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 \( p \in \mathbb{P} \), the projection of \( e \) to values with priority \( p \) is linearizable (w.r.t. SeqPQ). This property can be checked separately using register automata similar to the automata in [5] describing FIFO queue violations. This assumption excludes some obvious violations, such as an \( \text{rm}(a) \) operation happening before a \( \text{put}(a, p) \) operation, for some \( p \).

For \( \Gamma \in \{\text{UnmatchedMaxPriority}, \text{MatchedMaxPriority}\} \), we consider \( \Gamma\text{-complete} \) automata recognizing executions which contain only one maximal priority. This is w.l.o.g. 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 or equal to \( p \). Then,

\[ \Gamma\text{-linearizable} \iff e|_{\leq p} \text{ is } \Gamma\text{-linearizable for some maximal priority } p \text{ in } e. \]

\[ \text{Proof. (Sketch)} \]

For the “only-if” direction, let \( e \) be a data-differentiated execution linearizable w.r.t. \( l = u \cdot \text{put}(x, p) \cdot v \cdot \text{rm}(x) \cdot w \) s.t. \( \text{MatchedMaxPriority-Seq}(l, x) \) holds. Since the predicate \( \text{MatchedMaxPriority-Seq}(l, x) \) 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 predicate. 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 has a linearization \( l_2 \), then the execution has a linearization which is defined as an interleaving of \( l_1 \) and \( l_2 \).

Thus, let \( e \) be an execution such that \( e|_{\leq p} \) is linearizable w.r.t. \( l = u \cdot \text{put}(x, p) \cdot v \cdot \text{rm}(x) \cdot w \) where \( \text{MatchedMaxPriority-Seq}(l, x) \) holds. 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 \( \text{MatchedMaxPriority-Seq}(l, x) \) doesn’t constrain the values of priority incomparable to \( p \), we obtain that \( \text{MatchedMaxPriority-Seq}(l', \alpha) \) also holds.

The existence of \( \Gamma\text{-complete} \) automata enable an effective reduction of checking linearizability of concurrent priority queue implementations to state reachability. Section 4.2 discusses decidability results implied by this reduction.

\[ \text{Theorem 10. Let } I \text{ be a data-independent implementation. Then, there is a } \Gamma\text{-complete automaton } A(\Gamma) \text{ for each } \Gamma. \text{ Moreover, } I \subseteq \text{SeqPQ} \iff I \cap A(\Gamma) = \emptyset \text{ for all } \Gamma. \]
Checking Linearizability of Concurrent Priority Queues

Figure 2. An execution that is not MatchedMaxPriority completeness linearizable. We represent each operation as a time interval whose left, resp., right, bound corresponds to the call, resp., return action.

4.1 A MatchedMaxPriority-complete automaton

A differentiated execution $e$ is not MatchedMaxPriority-linearizable when all the put operations in $e$ using the maximal priority $p$ are matched, and $e$ is not linearizable w.r.t. the set of sequential executions satisfying 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 MatchedMaxPriority$^>$ the strengthening of MatchedMaxPriority with the condition that all the values other than $x$ have a priority strictly smaller than $p$ (corresponding to the first case), and by 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 \cite{15, 9, 18} whose states include only two registers, one for storing a priority guessed at the initial state, and one for storing the priority of the current action in the execution. The transitions can check equality or the order relation $\prec$ between the values stored in the two registers. Instead of formalizing the full class of register automata, we consider a simpler class which suffices our needs. Thus, we consider a class of labeled transition systems whose states consist of a finite control part and a register $r$ interpreted to elements of $\mathbb{P}$. The transition labels are:

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

The set of sequences (executions) accepted by such a transition system is defined as usual.

4.1.1 A MatchedMaxPriority$^>$-complete automaton

Figure 2 contains a typical example of an execution $e$ which is not MatchedMaxPriority$^>$-linearizable, where $p_1 \prec p_4$, $p_2 \prec p_4$, and $p_3 \prec p_4$. Intuitively, this is a violation because during the whole execution of $rm(b)$, the priority queue stores a smaller priority value (which should be removed before $b$). To be more precise, we define the interval of a value $x$ as the time interval from the return of a put $ret(put, x, p)$ to the call of the matching remove $call(rm, x)$, or to the end of the execution if such a call action doesn’t exist. This represents the time interval in which a value is guaranteed to be stored into the 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, the interval of each value of priority smaller than $p_4$ is pictured as a dashed line. There is no sequence $l$ s.t. $e \subseteq l$ and MatchedMaxPriority-Seq($l$, $b$) hold, since each time point from $call(rm, b)$ to $ret(rm, b)$ is included in the interval of a smaller priority value, and $rm(b)$ can’t take effect in the interval of a smaller priority value. To formalize this scenario we use the notion of left-right constraint defined below.
Definition 11. Let $e$ be a data-differentiated execution which contains only one maximal priority $p$, and only one value $x$ of priority $p$ (and no $rm(\text{empty})$ operations). The left-right constraint of $x$ is the graph $G$ where:
- the nodes are the values occurring in $e$,
- there is an edge from $d_1$ to $x$, if $\text{put}(d_1,\_)<_{hb} \text{put}(x, p)$ or $\text{put}(d_1,\_)<_{hb} \text{rm}(x)$,
- there is an edge from $x$ to $d_1$, if $\text{rm}(x)<_{hb} \text{rm}(d_1)$ or $\text{rm}(d_1)$ does not exists,
- there is an edge from $d_1$ to $d_2$, if $\text{put}(d_1,\_)<_{hb} \text{rm}(d_2,\_)$.

The execution in Figure 2 is not MatchedMaxPriority$^{\geq}$-linearizable because the left-right constraint of the maximal priority value $b$ contains the cycle $f \rightarrow d \rightarrow c \rightarrow b \rightarrow f$. The presence of such a cycle is equivalent to non MatchedMaxPriority$^{\geq}$-linearizability:

Lemma 12. Let $e$ be a data-differentiated execution such that Has-MatchedMaxPriority($c$) holds, $p$ is the maximal priority in $e$, and $\text{put}(x, p)$ and $\text{rm}(x)$ are only operations with arguments of priority $p$ in $e$. Then, $e$ is MatchedMaxPriority-linearizable iff the left-right constraint of $x$ contains no cycle going through $x$.

When the left-right constraint contains a cycle $d_1 \rightarrow \ldots \rightarrow d_m \rightarrow x \rightarrow d_1$, for some $d_1, \ldots, d_m \in \mathbb{D}$, we say that $x$ is covered by $d_1, \ldots, d_m$. The shape of an execution witnessing such a cycle (i.e., the alternation between call/return actions) can be identified 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 operations outside of the non-linearizable projection).

To define a MatchedMaxPriority$^{\geq}$-complete automaton, we need to consider all the possible orders between the call/return actions of the put/rm operations that add and respectively, remove the value $b$. The case where the put happens-before the remove (as in Figure 2) is pictured in Figure 4. This automaton captures the three possible ways of ordering the first action $\text{ret}(\text{put}, a, \_)$ w.r.t. the actions with value $b$, which are pictured in Figure 3(a) (this action cannot occur after $\text{call}(\text{rm}, b, \_)$ since $b$ must be covered by the $a$-s). The paths corresponding to these three possible orders are: $q_1 \rightarrow q_2 \rightarrow q_3 \ldots \rightarrow q_7$, $q_1 \rightarrow q_2 \rightarrow q_3 \ldots \rightarrow q_{10}$, and $q_1 \rightarrow q_{9} \rightarrow q_{10} \ldots \rightarrow q_7$. Figure 3 lists the four possible orderings of the call/return actions of adding and removing $b$, and also possible orders of the first $\text{ret}(\text{put}, a, \_)$ w.r.t. the actions with value $b$. Each such ordering corresponds to an automaton similar to the one in Figure 4, their union defining a MatchedMaxPriority$^{\geq}$-complete automaton.

### 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
checking linearizability of concurrent priority queues

Figure 4 A register automaton capturing the scenario in Figure 3(a). We use the following notations: \( A_1 = A \cup \{ \text{ret}(\text{rm}, a) \}, A_2 = A \cup \{ \text{call}(\text{put}, a, r) \}, A_3 = A_2 \cup \{ \text{ret}(\text{rm}, a) \} \), where 

\[
\begin{align*}
A &= \{ \text{call}(\text{put}, T, true), \text{ret}(\text{put}, T, true), \text{call}(\text{rm}, T), \text{ret}(\text{rm}, empty), \text{ret}(\text{rm}, empty) \}.
\end{align*}
\]

Figure 5 An execution that is not MatchedMaxPriority\(^{-}\)-linearizable.

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, where \( p_1 < p_4 \). 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 \( \text{put}(a, p_4) <_{pb} \text{put}(b, p_4) \) 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 only linearized in one of the two time intervals pictured with dotted lines in Figure 5. However, each of those time intervals ends before \( \text{call}(\text{rm}, a) \), and thus \( \text{rm}(a) \) cannot be linearized before \( \text{rm}(b) \).

To recognize the scenarios in Figure 5, we introduce an order \( <_{pb} \) between values which intuitively, can be thought of as “a value \( a \) is put before another value \( b \)”. More precisely, given a data-differentiated execution \( e \) and two values \( a \) and \( b \) of maximal priority, \( a <_{pb} b \) if one of the following holds: (1) \( \text{put}(a, \_\_) <_{hb} \text{put}(b, \_\_) \), (2) \( \text{rm}(a) <_{hb} \text{rm}(b) \), or (3) \( \text{rm}(a) <_{hb} \text{rm}(b, \_\_) \). Sometimes we use \( a <_{pb}^A b \), \( a <_{pb}^B b \), and \( a <_{pb}^C b \) to explicitly distinguish between these three cases. Let \( <_{pb}^* \) be the transitive closure of \( <_{pb} \).

To define the time intervals in which a remove like \( \text{rm}(b) \) in Figure 5, can be linearized (outside of intervals of smaller priority values) we use the notion of gap-point. As before, defining time intervals relies on an indexing of actions in an execution, starting with 0.

Definition 13. Let \( e \) be a data-differentiated execution with 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 \( x \) 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 not included in the interval of a value with priority smaller than \( p \).

The case of Figure 5 can be formally described as follows: \( a <_{pb}^A b \) 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\(^{-}\).
We omit the operations which can be ordered arbitrarily, e.g., put(b) in the cases (d) and (e).

**Figure 6** Orderings to be considered when defining a MatchedMaxPriority<sup>∞</sup>-complete automaton. We omit the operations which can be ordered arbitrarily, e.g., put(b) in the cases (d) and (e).

**Figure 7** A register automaton for the case in Figure 6(a), where \( A_1 = A \cup \{ \text{call}(\text{put}, c, r) \} \), \( A_2 = A_1 \cup \{ \text{ret}(\text{put}, b, r) \} \), \( A_3 = A_2 \cup \{ \text{ret}(\text{rm}, c) \} \), \( A_4 = A_3 \cup \{ \text{ret}(\text{put}, b, r), \text{ret}(\text{rm}, c) \} \), where \( A = \{ \text{call}(\text{put}, T, true), \text{ret}(\text{put}, T, true), \text{call}(\text{rm}, T), \text{ret}(\text{rm}, T), \text{call}(\text{rm}, empty), \text{ret}(\text{rm}, empty) \} \).

**Lemma 14.** Let \( e \) be a data-differentiated execution with only one maximal priority \( p \) such that \( \text{Has-MatchedMaxPriority}(e) \) holds. Then, \( e \) is not MatchedMaxPriority<sup>∞</sup>-linearizable iff \( e \) contains two values \( x \) and \( y \) of maximal priority \( p \) such that \( y <_{\text{rb}} x \), and the rightmost gap-point of \( x \) is strictly smaller than the index of \( \text{call}(\text{put}, y, p) \) or \( \text{call}(\text{rm}, y) \).

The following shows that the number of values needed to witness that \( y <_{\text{rb}} x \), for some \( x \) and \( y \), is bounded.

**Lemma 15.** Let \( e \) be a data-differentiated execution such that \( a <_{\text{rb}} a_1 <_{\text{rb}} \ldots <_{\text{rb}} a_m <_{\text{rb}} b \) holds for some set of values \( a, a_1, \ldots, a_m, b \). Then, one of the following holds:

- \( a <_{\text{rb}} b, a <_{\text{rb}} b_1, \) or \( a <_{\text{rb}} b_2 \),
- \( a <_{\text{rb}} a_1 <_{\text{rb}} b \) or \( a <_{\text{rb}} a_1 <_{\text{rb}} b, \) for some \( i \).

To characterize violations to MatchedMaxPriority<sup>∞</sup>-linearizability, one has to consider all the possible orders between call/return actions of the operations on values \( a, b, \) and \( a_i \) in Lemma 15, and the rightmost gap point of \( b \). Excluding the inconsistent cases, we are left with the five orders in Figure 6, where \( \circ \) denotes the rightmost gap point of \( b \). For each case, we define an automaton recognizing the induced set of violations. The register automaton for the case in Figure 6(a) is shown in Figure 7. In this case, Lemma 14 is 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 (thus, there is no gap point of \( b \) which occurs after \( \text{call}(\text{rm}, a) \)). By data-independence, these lower priority values can be renamed to a fixed value \( c \).
4.2 Decidability Result

We describe a class $C$ of data-independent implementations for which linearizability w.r.t. $\text{SeqPQ}$ is decidable. The implementations in $C$ allow an unbounded number of values but a bounded number of priorities. Each method has a finite set of local variables storing Boolean values or values from $D$. Methods communicate through a finite number of shared variables interpreted also as Booleans or values from $D$. To ensure data independence, values in $D$ may be assigned, but never used in Boolean expressions (e.g., of if-then-else statements). This class captures typical implementations, or finite-state abstractions thereof, e.g., obtained via predicate abstraction. The $\Gamma$-complete automata we define use a fixed set $D = \{a, b, c, a_1, \top\}$ of values ($a_1$ is needed to deal with the second item in Lemma 15). Therefore, for any $\Gamma$, $C \cap A(\Gamma) \neq \emptyset$ iff $C_D \cap A(\Gamma) \neq \emptyset$, 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 System with States (VASS). Since the set of values and priorities is bounded, each method invocation can be represented by a finite-state automaton (see [4]). For a fixed set of priorities $P \subseteq P$, the register automata $A(\Gamma)$ can be transformed to finite-state automata (the number of valuations of the registers is bounded). Thus, checking linearizability of an implementation in $C$ is PSPACE when the number of threads is bounded, and EXPSPACE, otherwise. Moreover, reachability in VASSs can be reduced to checking linearizability of such an implementation. Essentially, given an instance of the VASS reachability problem, one can define a priority queue implementation where the $\text{put}$ methods behave correctly and additionally, they include the code of the VASS simulation defined in [4], and the $\text{rm}$ methods behave correctly, except for the moment where the target state is reached, in which case they trigger a linearizability violation by returning an arbitrary value.

$\triangleright$ Theorem 16. Verifying whether an implementation in $C$ is linearizable w.r.t. $\text{SeqPQ}$ 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 are also based on reductions to safety verification, e.g., [1, 13, 21]. The approach in [21] considers implementations where operations’ linearization points are manually specified. Essentially, this approach instruments the implementation with ghost variables simulating the ADT specification at linearization points. This approach is incomplete since not all implementations have fixed linearization points. Aspect-oriented proofs [13] reduce linearizability to the verification of four simpler safety properties. This approach has only been applied to queues, and has not produced a fully automated and complete proof technique. The work in [10] proves linearizability of stack implementations with an automated proof assistant. Their approach does not lead to full automation however, e.g., by reduction to safety verification.

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

References


Checking Linearizability of Concurrent Priority Queues


