Quantitative Relaxation of Concurrent Data Structures*

Thomas A. Henzinger Ali Sezgin
IST Austria
{tah,asezgin}@ist.ac.at

Christoph M. Kirsch Hannes Payer Ana Sokolova
University of Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

Abstract. There is a trade-off between performance and correctness in implementing concurrent data structures. Better performance may be achieved at the expense of relaxing correctness, by redefining the semantics of data structures. We address such a redefinition of data structure semantics and present a systematic and formal framework for obtaining new data structures by quantitatively relaxing existing ones. We view a data structure as a sequential specification S containing all "legal" sequences over an alphabet Σ of method calls. Relaxing the data structure corresponds to defining a distance from *any* sequence over Σ to the sequential specification: the relaxed sequential specification S_k contains all sequences over Σ within distance k from S. In contrast to other existing work, our relaxations are semantic (distance in terms of data structure states). As an instantiation of our framework, we present a simple yet generic relaxation scheme along with several ways of computing distances. We show that this relaxation, when further instantiated on queues, stacks, and priority queues, amounts to tolerating bounded out-of-order behavior, which cannot be captured by a purely syntactic relaxation (distance in terms of sequence manipulation, e.g. edit distance). We also present various concurrent implementations of queue, stack, and priority queue relaxations and argue that bounded relaxations provide the means for trading correctness for performance in a controlled way. The relaxations are monotonic which further highlights the trade-off: increasing k increases the number of permitted sequences, which increases the performance.

1 Introduction

Concurrent data structures may be a performance and scalability bottleneck and thus prevent effective use of increasingly parallel hardware [11]. There is a trade-off between scalability (performance) and correctness in implementing concurrent data structures. A remedy to the scalability problem is to relax the semantics of concurrent data structures. The semantics is given by some notion of equivalence with sequential behavior. The equivalence is determined by a consistency condition, most commonly linearizability [5], and the sequential behavior is inherited from the sequential version of the data structure (e.g., the sequential behavior of a concurrent stack is a regular stack). Therefore, relaxing the semantics of a concurrent data structure amounts to either weakening the consistency condition (linearizability being replaced with sequential consistency or quiescent consistency) or redefining (relaxing) its sequential specification. In this paper, we present a framework for relaxing the sequential specification in a quantitative manner.

For an example of a relaxation, imagine a k-stack in which each `pop` removes one of the most recent k elements and an operation `size` which returns a value that is k away from the correct size. It is intuitively clear that such a k-stack relaxes a regular stack, but current theory does not provide means to quantify the relaxation. Our framework does, it provides a way to formally describe and quantitatively assess such relaxations.

* This work has been supported by the Austrian Science Fund (Elise Richter Fellowship LEGO-CPT, V00125, and RiSE NFN on Rigorous Systems Engineering, S11402-N23 and S11404-N23), the European Research Council (ERC) Advanced Grant QUAREM (Quantitative Reactive Modeling), and the National Science Foundation (CNS1136141).

We view a data structure as a sequential specification S consisting of all semantically correct sequences of method calls. We identify the sequential specification with a particular labeled transition system (LTS) whose states are sets of sequences in S with indistinguishable future behavior and transitions are labeled by method calls. A sequence is in the sequential specification if and only if it is a finite trace of this LTS.

Our framework for quantitative relaxation of concurrent data structures amounts to specifying costs of transitions and paths. In the LTS, only *correct* transitions are allowed, e.g., a transition labeled by `pop(a)` is only possible in a state of a stack with `a` as top element. In a relaxation, we are exactly interested in allowing the *wrong* transitions, but they will have to incur cost. Our framework makes this possible in a controlled quantitative way.

The framework is instantiated through specifying two cost functions: A local function, *transition cost*, that assigns a penalty to each wrong transition, and a global function, *path cost*, that accumulates the local costs (using, e.g., maximum, sum, or average) to obtain the overall distance of a sequence. Via this local-global dichotomy, we are able to achieve a separation of concerns, modularity and flexibility: Different transition costs can be used with the same path cost, or vice versa, leading to different relaxations. Once the distance of a sequence from the original sequential specification S is defined in this way, a k-relaxation of the data structure becomes the set of all sequences within distance k from S.

Returning to the stack example above, we can set the transition cost of a `pop` transition at a state to be the number of elements that are between the popped element and the top of the stack. We can define the path cost to be the maximum transition cost that occurs along a sequence. Then, the corresponding k-relaxation precisely captures what we intuitively described.

We instantiate the framework on two levels. On the generic level, we present a generic relaxation, called out-of-order relaxation, which provides a way to assign transition costs, together with several different path cost functions for *any* data structure. On the concrete level, we instantiate the out-of-order relaxation to queues, stacks, and priority queues. We spell out the effects of the relaxation in each of these concrete cases and show that they indeed correspond to the intuitive idea of bounded relaxed out-of-order behavior.

We also give prototype implementations of the out-of-order relaxations of several concurrent data structures, such as queues, stacks, and priority queues. Our experimental results demonstrate that these implementations decrease contention. We conclude the paper with a brief survey of related work, putting special emphasis on quasi-linearizability [1], the only other work we are aware of that also tackled the problem of quantitatively relaxing sequential data structures for better performance in the concurrent setting. As opposed to our semantic (state-based) approach in assigning distances to sequences, the relaxation of [1] is syntactic (permutation-based). In the final section we argue that (1) the semantic approach is more expressive than the syntactic one, and (2) it allows the designer of a data structure to formally capture the intent of a specific relaxation more easily and naturaly.

To conclude, the main contribution of this paper is the framework for quantitative relaxations of data structures. The way to the framework is paved by formally capturing the semantics of a data structure. Other contributions made possible by the framework are: the generic out-of-order relaxation of data structures; characterizations of the out-of-order relaxation in concrete terms for queues, stacks, and priority queues; and demonstration of the positive effect of relaxations on scalability via several prototype implementations obtained by out-of-order relaxation.

2 Data structures, specifications, states

Let Σ be a set of methods including input and output values. We will refer to Σ as the sequential alphabet. A *sequential history* $\mathbf{s}$ is an element of Σ^*, i.e., a sequence over Σ. A *data structure* is a *sequential specification* S which is a prefix-closed set of sequential histories, $S \subseteq \Sigma^*$.

The following definition is the core of our way of capturing semantics. Let S be a sequential specification.

Definition 21 *Two sequential histories* $\mathbf{s}, \mathbf{t} \in S$ *are* S-equivalent*, written* $\mathbf{s} \equiv_S \mathbf{t}$*, if for any sequence* $\mathbf{u} \in \Sigma^*$*,* $\mathbf{su} \in S$ *if and only if* $\mathbf{tu} \in S$*.*

It is clear that $\equiv_S$ is an equivalence relation. By $[\mathbf{s}]_S$ we denote the S-equivalence class of $\mathbf{s}$. Intuitively, two sequences in the sequential specification are S-equivalent if they lead to the same "state". The following simple property follows directly from the definition of S-equivalence.

Lemma 22 *If* $\mathbf{s} \equiv_S \mathbf{t}$ *and* $\mathbf{su} \in S$*, then* $\mathbf{su} \equiv_S \mathbf{tu}$*.*

The intuition about states is made explicit in the next definition. In addition, we point out particular "minimal" representatives of a state.

Definition 23 *A state of a data structure with sequential specification* S *is an equivalence class* $[\mathbf{s}]_S$ *with respect to* $\equiv_S$*. For a state* $q = [\mathbf{s}]_S$*, the kernel of* q *is the set*

$$\ker(q) = \{\, \mathbf{t} \in [\mathbf{s}]_S \mid \mathbf{t} \text{ has minimal length } \}.$$

A sequence $\mathbf{s} \in S$ *is a kernel sequence if* $\mathbf{s} \in \ker([\mathbf{s}]_S)$*.*

A data structure determines a labeled transition system (LTS) that we define next.

Definition 24 *Let* S *be a (sequential specification of a) data structure. Its corresponding LTS is* $\mathrm{LTS}(S) = (Q, \Sigma, \rightarrow, q_0)$ *with set of states* $Q = S/\equiv_S = \{\, [\mathbf{s}]_S \mid \mathbf{s} \in S \}$*, set of labels* Σ*, transition relation* $\rightarrow \subseteq Q \times \Sigma \times Q$ *given by* $[\mathbf{s}]_S \xrightarrow{m} [\mathbf{s}m]_S$ *if and only if* $\mathbf{s}m \in S$*, and initial state* $q_0 = [\varepsilon]_S$*.*

Note that the transition relation is well defined (independent of the choice of a representative) due to Lemma 22. Also q_0 is well defined since S is prefix closed. We write $q \xrightarrow{m}$ if there is an m-labeled transition from q to some state; $q \overset{m}{\not\rightarrow}$ if there is no m-labeled transition from q. We also write $q \xrightarrow{\mathbf{u}}$ if there is a $\mathbf{u}$-labeled path of transitions starting from q, and $q \overset{\mathbf{u}}{\not\rightarrow}$ if it is not the case that $q \xrightarrow{\mathbf{u}}$. The following immediate observation provides the exact correspondence between the sequential specification of a data structure and its LTS: S is the set of finite traces of the initial state of $LTS(S)$.

Lemma 25 *Let* S *be a sequential specification with* $\mathrm{LTS}(S) = (Q, \Sigma, \rightarrow, q_0)$*. Then for any* $\mathbf{u} \in \Sigma^*$ *we have* $\mathbf{u} \in S$ *if and only if* $q_0 \xrightarrow{\mathbf{u}}$*.*

Notes on concurrency. Let $\Sigma_{ir} = \Sigma_i \cup \Sigma_r$ with $\Sigma_i = \{m_i \mid m \in \Sigma\}$ and $\Sigma_r = \{m_r \mid m \in \Sigma\}$ be the *concurrent alphabet* in which for every method $m \in \Sigma$ we distinguish between its *invocation event* m_i and its *response event* m_r. A *concurrent history* $\mathbf{c}$ is an element of ${\Sigma_{ir}}^*$ with the property that if m_r appears in $\mathbf{c}$ then m_i also appears in c and it does so before m_r.

Every sequential history "is" a concurrent history as well, i.e., we identify a sequential history $\mathbf{s} \in \Sigma^*$ with the concurrent history $\widehat{\mathbf{s}} \in {\Sigma_{ir}}^*$, defined inductively by

$$\widehat{\varepsilon} = \varepsilon \quad \text{and} \quad \widehat{m\mathbf{t}} = m_i m_r \widehat{\mathbf{t}} \quad \text{for } m \in \Sigma \text{ and } \mathbf{t} \in \Sigma^*.$$

A concurrent history $\mathbf{c}$ is *linearizable* with respect to sequential specification S if there exists a permutation $\widehat{\mathbf{s}}$ of $\mathbf{c}$ such that $\mathbf{s} \in S$ is a sequential history, and for any two methods m and n, if m_r precedes n_i in $\mathbf{c}$, then m precedes n in s.[1]

In this paper, we are mainly concerned with the sequential specification of a data structure. By relaxing it we get a new data structure, a relaxed version of the original one. The relaxation has applications on the concurrent side: relaxed data structures may allow for better-performing concurrent implementations.

[1] For simplicity, we avoid a detailed definition of linearizability here; see [5] for a formal treatment.

Remark 26 A relaxation S_k of a data structure S directly leads to a relaxed notion of linearizability: a concurrent history $\mathbf{c}$ is k-linearizable if it is linearizable with respect to the k-relaxed specification S_k. It is important to mention here that the actual condition of linearizability does not change, only the sequential specification does. This approach was used for defining quasi-linearizability [1] and k-linearizability of FIFO queues [7].

We discuss concurrent implementations of proposed relaxations in Section 6. Until then, we deal with the sequential specification only.

3 Framework for quantitative relaxations

In this section we present our framework for quantitatively relaxing data structures. Let $S \subseteq \Sigma^*$ be a data structure with $\mathrm{LTS}(S) = (Q, \Sigma, \to, q_0)$. Our goal is to relax S to a so-called k-relaxed specification $S_k \subseteq \Sigma^*$ in a bounded way, with k providing the bound.

Giving a relaxation for a data structure S amounts to the following three steps:

1. **Completion.** From $\mathrm{LTS}(S) = (Q, \Sigma, \to, q_0)$ we construct the completed labeled transition system $\mathrm{LTS}_c(S) = (Q, \Sigma, Q \times \Sigma \times Q, q_0)$ with transitions from any state to any other state by any method.
2. **Transition costs.** From $\mathrm{LTS}_c(S)$ a quantitative labeled transition system $\mathrm{QLTS}(S) = (Q, \Sigma, Q \times \Sigma \times Q, q_0, C, \mathrm{cost})$ is constructed. Here C is a well-ordered cost domain, hence it has a minumum that we denote by 0, and $\mathrm{cost}\colon Q \times \Sigma \times Q \to C$ is the transition cost function satisfying

$$\mathrm{cost}(q, m, q') = 0 \quad \text{if and only if} \quad q \xrightarrow{m} q' \text{ in } \mathrm{LTS}(S).$$

   We write $q \xrightarrow{m,k} q'$ for the quantitative transition with $\mathrm{cost}(q, m, q') = k$. A quantitative path of $\mathrm{QLTS}(S)$ is a sequence

$$\kappa = q_1 \xrightarrow{m_1,k_1} q_2 \xrightarrow{m_2,k_2} q_3 \ldots q_n \xrightarrow{m_n,k_n} q_{n+1}.$$

   The sequence $\tau = (m_1, k_1)(m_2, k_2)\ldots(m_n, k_n) \in (\Sigma \times C)^*$ is the quantitative trace of κ, notation $\mathrm{qtr}(\kappa)$ and the sequence $\mathbf{u} = m_1 \ldots m_n$ is the trace of the quantitative path κ and of the quantitative trace $\mathrm{qtr}(\kappa)$, notation $\mathrm{tr}(\kappa) = \mathrm{tr}(\mathrm{qtr}(\kappa)) = \mathbf{u}$. By $\mathrm{qtr}(\mathbf{u})$ we denote the set of all quantitative traces of quantitative paths starting in the initial state with trace $\mathbf{u}$ and by $\mathrm{qtr}(S)$ the set of all quantitative traces of quantitative paths starting in the initial state.
3. **Path cost function.** We choose a monotone path cost function $\mathrm{pcost}\colon \mathrm{qtr}(S) \to C$. Monotonicity here is with respect to prefix order: if a quantitative trace τ is a prefix of a quantitative trace τ', then $\mathrm{pcost}(\tau) \leqslant \mathrm{pcost}(\tau')$.

Having performed these three steps, we can define the k-relaxed specification.

Definition 31 *The k-relaxed specification S_k for $k \in C$ contains all sequences that have a distance at most k from S,*

$$S_k = \{\mathbf{u} \in \Sigma^* \mid d_S(\mathbf{u}) \leqslant k\}$$

where $d_S(\mathbf{u})$ is the distance of $\mathbf{u}$ to the sequential specification S given by

$$d_S(\mathbf{u}) = \min\{\mathrm{pcost}(\tau) | \tau \in \mathrm{qtr}(\mathbf{u})\}.$$

Remark 32 Both the distance d_S and the relaxed specification S_k are actually parametric in the transition cost function as well as in the path cost function. For simplicity, we prefer a light, overloaded notation that does not explicitly mention these parameters.

Some obvious properties of the quantified relaxations resulting from our framework are:

– $S_0 = S$, ensured by the condition on the transition cost function.
– Every relaxation S_k is prefix closed, ensured by the monotonicity of the path cost function.
– The relaxations are monotone, i.e., if $k \leqslant m$, then $S_k \subseteq S_m$.

To conclude, in order to relax a data structure all that one needs is a cost domain C, a transition cost for each transition in the completed LTS (item 2. above), and a path cost function (item 3. above).

4 Generic out-of-order relaxation

In this section we illustrate the relaxation framework on one generic example. The value and generality of this particular example becomes evident in Section 5 when we instantiate it to concrete data structures. Let $S \subseteq \Sigma^*$ be a data structure with $\mathrm{LTS}(S) = (Q, \Sigma, \to, q_0)$. We first fix the cost domain to $C = \mathbb{N} \cup \{\infty\}$. Next, we define a transition cost function $\mathit{scost}\colon Q \times \Sigma \times Q \to C$, called *segment cost*, and mention two other related transition cost functions. Finally, we suggest several path cost functions to be used along with the segment cost.

Definition 41 *Let $t = (q, m, q')$ be a transition in $\mathrm{LTS}_c(S)$. Let $\mathbf{v}$ be a sequence with minimal length satisfying one of the following two conditions:*

(1) There exist sequences $\mathbf{u}$, $\mathbf{w}$ such that $\mathbf{uvw} \in \ker(q)$ and $\mathbf{uw}$ is a kernel sequence and either
  (i) $[\mathbf{uw}]_S \xrightarrow{m} [\mathbf{u'w}]_S$ and $q' = [\mathbf{u'vw}]_S$, or
  (ii) $[\mathbf{uw}]_S \xrightarrow{m} [\mathbf{uw'}]_S$ and $q' = [\mathbf{uvw'}]_S$.
(2) There exist sequences $\mathbf{u}$, $\mathbf{w}$ such that $\mathbf{uw} \in \ker(q)$ and $\mathbf{uvw}$ is a kernel sequence and either
  (i) $[\mathbf{uvw}]_S \xrightarrow{m} [\mathbf{u'vw}]_S$ and $q' = [\mathbf{u'w}]_S$, or
  (ii) $[\mathbf{uvw}]_S \xrightarrow{m} [\mathbf{uvw'}]_S$ and $q' = [\mathbf{uw'}]_S$.

Then the segment cost is given by the length of $\mathbf{v}$, $\mathit{scost}(t) = |\mathbf{v}|$. If such a sequence $\mathbf{v}$ does not exist for t, then $\mathit{scost}(t) = \infty$.

Intuitively, segment cost of a relaxed transition is the length of the shortest subword ($\mathbf{v}$) whose removal (1) or insertion (2) into the kernel sequence enables a transition. Observe that the transition can be taken in $\mathrm{LTS}(S)$ if and only if its segment cost is 0, obtained by setting $\mathbf{v} = \varepsilon$. We will see in the next section that this cost quantifies *out-of-order* updates or observations, such as removing an element other than the head of a queue or returning an element other than the top element in a stack. There are two specializations of the segment cost that we isolate in the next definition.

Definition 42 Discarding cost *is the transition cost function defined by Definition 41 when condition (2) is removed.* Filling cost *is the transition cost function defined by Definition 41 when condition (1) is removed.*

Let $S \subseteq \Sigma^*$ be a data structure, $\mathbf{u}$ a sequence over Σ, and $\tau = (m_1, k_1)(m_2, k_2)\dots(m_n, k_n)$ a quantitative trace in $\mathrm{qtr}(S)$ such that $\mathrm{tr}(\tau) = \mathbf{u}$. We define the following path cost functions:

– The *maximal cost*, $\mathrm{pcost}_{max}\colon \mathrm{qtr}(S) \to \mathbb{N} \cup \{\infty\}$, maps τ to the maximal transition cost along it. Formally, $\mathrm{pcost}_{max}(\tau) = \max\{k_i \mid 1 \leqslant i \leqslant n\}$.
– The *L-interval cost*, $\mathrm{pcost}_L\colon \mathrm{qtr}(S) \to \mathbb{N} \cup \{\infty\}$, for $L \subseteq \Sigma$ maps τ to the maximum number of L-labeled transitions in a subpath of τ which contains no L-labeled transitions with transition cost 0. Formally, let $\mathbf{v} = \mathbf{u}|L = m_{i_1} \dots m_{i_l}$ be the projection of $\mathbf{u}$ onto the sub-alphabet L. Then we have

$$\mathrm{pcost}_L(\tau) = \max\{k - j + 1 \mid k_{i_r} \neq 0 \text{ for } 1 \leqslant j \leqslant r \leqslant k \leqslant l\}.$$

– The *L-interval restricted maximal cost*, $\mathrm{pcost}_{max|L}\colon \mathrm{qtr}(S) \to \mathbb{N} \cup \{\infty\}$, for $L \subseteq \Sigma$ maps τ to the maximum of the sum of transition cost and the number of L-labeled transitions since the last L-labeled transition with cost 0. Formally, let again $\mathbf{v} = \mathbf{u}|L = m_{i_1} \dots m_{i_l}$ be the projection of $\mathbf{u}$ onto the sub-alphabet L. For $1 \leqslant i \leqslant n$, let $l_i = \max\{r - s + 1 \mid k_{i_t} \neq 0 \text{ for } 1 \leqslant s \leqslant t \leqslant r\}$ if $m_i \in L, i = i_r$ for some $1 \leqslant r \leqslant l$, and $l_i = 0$ otherwise. Then $\mathrm{pcost}_{max|L}(\tau) = \max\{k_i + l_i \mid 1 \leqslant i \leqslant n\}$.

Remark 43 The maximal cost is certainly intuitive and easily understandable. The other two path cost functions may appear to be less intuitive at first sight. We present them here for two reasons: (1) they illustrate more complex examples within the variety of possible path cost functions, and (2) when instantiating the generic out-of-order relaxation to concrete data structures (Section 5) they produce intuitively meaningful path costs.

5 Quantitatively relaxed queues, stacks, and priority queues

In this section we apply the relaxation of Section 4 to FIFO queues, stacks, and priority queues[2]. In order to show the full generality of the out-of-order relaxation, we consider additional observer methods like `head`, `top`, `max`, respectively, and `size`.

FIFO queue. The set of methods for a FIFO queue, with data set D and $D_{\mathtt{null}} = D \cup \{\mathtt{null}\}$, is

$$\Sigma_Q = \{\mathtt{enq(d)} \mid \mathtt{d} \in D\} \cup \{\mathtt{deq(d)} \mid \mathtt{d} \in D_{\mathtt{null}}\} \cup \{\mathtt{head(d)} \mid \mathtt{d} \in D_{\mathtt{null}}\} \cup \{\mathtt{size(n)} \mid \mathtt{n} \in \mathbb{N}\}.$$

The sequential specification S_Q consists of all queue-valid sequences, i.e., sequences in which each `deq` deques the head of the queue, each `enq` enqueues at the tail of the queue, each `head` observes the head element, and `size` returns the current size of the queue. For instance, the following sequence $\mathbf{s}_Q = \mathtt{enq(a)enq(b)deq(a)}$ is in the sequential specification S_Q, whereas the sequence $\mathbf{t}_Q = \mathtt{enq(a)enq(b)deq(b)}$ is not.

One can easily show that kernel sequences of a FIFO queue are all sequences $\mathbf{s} \in \{\mathtt{enq(d)} \mid \mathtt{d} \in D\}^*$. Moreover, for any state $q = [\mathbf{s}]_{S_Q}$ of the FIFO queue, there is a unique sequence in $\ker(q)$, i.e., $|\ker(q)| = 1$. This implies that different sequences in $\mathbf{s} \in \{\mathtt{enq(d)} \mid \mathtt{d} \in D\}^*$ represent different states. As a consequence, the transition relation of $\mathrm{LTS}(S_Q)$ can be described in a concise way. Let $\mathbf{s}$ be a kernel sequence of a queue. We have, for the updater methods,

$$[\mathbf{s}]_{S_Q} \xrightarrow{\mathtt{enq(a)}} [\mathbf{s} \cdot \mathtt{enq(a)}]_{S_Q}, [\mathbf{s}]_{S_Q} \xrightarrow{\mathtt{deq(a)}} [\mathbf{s}']_{S_Q} \text{ if } \mathbf{s} = \mathtt{enq(a)} \cdot \mathbf{s}', \text{ and } [\mathbf{s}]_{S_Q} \xrightarrow{\mathtt{deq(null)}} [\varepsilon]_{S_Q} \text{ if } \mathbf{s} = \varepsilon$$

and for the observer methods

$$[\mathbf{s}]_{S_Q} \xrightarrow{\mathtt{head(a)}} [\mathbf{s}]_{S_Q} \text{ if } \mathbf{s} = \mathtt{enq(a)} \cdot \mathbf{s}', [\mathbf{s}]_{S_Q} \xrightarrow{\mathtt{head(null)}} [\mathbf{s}]_{S_Q} \text{ if } \mathbf{s} = \varepsilon, \text{ and } [\mathbf{s}]_{S_Q} \xrightarrow{\mathtt{size(n)}} [\mathbf{s}]_{S_Q} \text{ if } |\mathbf{s}| = \mathtt{n}.$$

Let $\mathbf{s}$ be a kernel sequence. A kernel sequence $\mathbf{s}'$ is

– `enq(a)`-out-of-order-k from $\mathbf{s}$ if $\mathbf{s}' = \mathbf{u} \cdot \mathtt{enq(a)} \cdot \mathbf{v}$ where $\mathbf{s} = \mathbf{uv}$, $\mathbf{v}$ is minimal, and $|\mathbf{v}| = k$;
– `deq(a)`-out-of-order-k from $\mathbf{s}$ if $\mathbf{s}' = \mathbf{vu}$ where $\mathbf{s} = \mathbf{v} \cdot \mathtt{enq(a)} \cdot \mathbf{u}$, $\mathbf{v}$ is minimal, and $|\mathbf{v}| = k$;
– `deq(null)`-out-of-order-k from ε iff $|\mathbf{s}'| = k$;
– `head(a)`-out-of-order-k (from itself) if $\mathbf{s}' = \mathbf{v} \cdot \mathtt{enq(a)} \cdot \mathbf{u}$, $\mathbf{v}$ is minimal, and $|\mathbf{v}| = k$;
– `head(null)`-out-of-order-k (from itself) if $|\mathbf{s}'| = k$;
– `size(n)`-out-of-order-k (from itself) if $||\mathbf{s}'| - \mathtt{n}| = k$.

[2] In Appendix C, we present another example (relaxed shared counter) that instantiates the quantitative relaxation framework.

Having these definitions in place, it is not difficult to show the following result, which describes the effect of the generic out-of-order relaxation on a FIFO queue.

Proposition 51 *Let* $\mathbf{s}$ *and* $\mathbf{s}'$ *be two kernel sequences of a FIFO queue. Then* $[\mathbf{s}]_{S_Q} \xrightarrow{m,k} [\mathbf{s}']_{S_Q}$ *in the out-of-order relaxation with segment cost if and only if* $\mathbf{s}'$ *is m-out-of-order-k from* $\mathbf{s}$.

From the proof of Proposition 51 (Appendix A.1) it is evident that for `enq`, `deq`, and `head` methods, discard cost suffices. For `size` methods, we need also filling cost.

Finally, let us mention that the relaxations can be applied method-wise. In Section 6 we implement k-relaxed queues with only `enq` and `deq` methods, of which only `deq` is relaxed. We specify these relaxed queues in the next definition.

Definition 52 *We name three versions of k-relaxations of FIFO queues.*

- *The* out-of-order k-FIFO queue *is obtained using segment cost for* `deq`*-transitions and maximal cost. In such a k-FIFO queue, each* `deq` *dequeues an element that is at most k away from the head.*
- *The* lateness k-FIFO queue *is obtained using segment cost and* `deq`*-interval cost. In the lateness k-FIFO queue at most the k-th consecutive* `deq` *dequeues the head.*
- *The* restricted out-of-order k-FIFO queue *is obtained using the segment cost and* `deq`*-interval restricted maximal cost. In the restricted out-of-order k-FIFO queue, each* `deq` *removes an element at most* $k - l$ *away from the head, where* l *is the current lateness of the head.*

Stack. Here we describe the out-of-order relaxation of a stack. For simplicity, we focus only on push and pop methods. A complete treatment of stack relaxations (including observer methods) can be found in Appendix A.2. The set of methods of a stack, with data in a set D, is

$$\Sigma_S = \{\texttt{push(d)} \mid \texttt{d} \in D\} \cup \{\texttt{pop(d)} \mid \texttt{d} \in D \cup \{\texttt{null}\}\}.$$

The sequential specification S_S consists of all stack-valid sequences, i.e., sequences in which each `pop` pops the top of the stack and each `push` pushes an element at the top.

Kernel sequences of a stack are all sequences $\mathbf{s} \in \{\texttt{push(d)} \mid d \in D\}^*$. Moreover, for any state q we again have $|\ker(q)| = 1$. Therefore, the transitions of $LTS(S_S)$ are fully described by

$$[\mathbf{s}]_{S_S} \xrightarrow{\texttt{push(a)}} [\mathbf{s} \cdot \texttt{push(a)}]_{S_S}, [\mathbf{s}]_{S_S} \xrightarrow{\texttt{pop(a)}} [\mathbf{s}']_{S_S} \text{ if } \mathbf{s} = \mathbf{s}' \cdot \texttt{push(a)}, \text{ and } [\mathbf{s}]_{S_S} \xrightarrow{\texttt{pop(null)}} [\varepsilon]_{S_S} \text{ if } \mathbf{s} = \varepsilon.$$

In a similar way as for FIFO queue, we can define when a kernel sequence is m-out-of-order-k from another kernel sequence, for m being a stack method. The analogue of Proposition 51 (obtained by replacing "FIFO queue" by "stack") holds for a stack as well, resulting in analogous stack relaxations.

Priority queue. The data set of a priority queue needs to be well-ordered, since data items carry priority as well. We take the data set to be $\mathbb{N}$. The smaller the number, the higher the priority. As for a stack, we present only the core of the out-of-order relaxation for priority queue, more details (also for observer methods) can be found in Appendix A.3. The set of methods is

$$\Sigma_P = \{\texttt{ins(n)} \mid \texttt{n} \in \mathbb{N}\} \cup \{\texttt{rem(n)} \mid \texttt{n} \in \mathbb{N} \cup \{\texttt{null}\}\}.$$

The sequential specification S_P consists of all priority-queue-valid sequences, i.e., sequences in which each `rem` removes an element with highest available priority.

Kernel sequences of a priority queue are all sequences $\mathbf{s} \in \{\texttt{ins(n)} \mid \texttt{n} \in \mathbb{N}\}^*$. Unlike for queue and stack, there may be more than one sequence representing a state. For a state q, if $\mathbf{s} \in \ker(q)$, then also any permutation of $\mathbf{s}$ is in $\ker(q)$. Nevertheless, the order provides a canonical representative of a state: the

unique kernel sequence ordered in non-decreasing priority[3]. Let $\mathbf{s}$ be a canonical kernel sequence. The transitions of $LTS(S_{\mathcal{P}})$ are fully described by

$$[\mathbf{s}]_{S_{\mathcal{P}}} \xrightarrow{\texttt{ins(n)}} [\mathbf{s}\cdot\texttt{ins(n)}]_{S_{\mathcal{P}}}, [\mathbf{s}]_{S_{\mathcal{P}}} \xrightarrow{\texttt{rem(n)}} [\mathbf{s}']_{S_{\mathcal{S}}} \text{ if } \mathbf{s} = \texttt{ins(n)}\cdot\mathbf{s}', \text{ and } [\mathbf{s}]_{S_{\mathcal{S}}} \xrightarrow{\texttt{rem(null)}} [\varepsilon]_{S_{\mathcal{S}}} \text{ if } \mathbf{s} = \varepsilon.$$

Again, we define when a kernel sequence is m-out-of-order-k from another kernel sequence, where m is a priority queue method. Then the analogue of Proposition 51 (obtained by replacing "FIFO queue" by "priority queue") holds as well, resulting in analogous priority queue relaxations.

6 Implementations of relaxed data structures

In this section, we examine k-relaxed versions of a concurrent FIFO queue (k-FIFO queue). We present prototype implementations with restricted out-of-order relaxation, out-of-order relaxation, and lateness relaxation of the dequeue operation. We have also implemented a k-stack and a k-priority queue, based on Treiber's lock-free stack [12] and a skiplist-style priority queue [8], respectively. The details of the implementations and experiments are similar to the k-FIFO implementations but omitted due to lack of space.

k-FIFO queue. Our prototype implementation of a k-FIFO queue is based on the lock-free Michael-Scott FIFO queue (MS) [9], which uses a singly-linked list as queue representation. With MS the enqueue operation always operates on the tail pointer using a compare-and-swap (CAS) operation and the dequeue operation always operates on the head pointer using a CAS operation. In a high-contention scenario where multiple threads work concurrently and in parallel on the queue these pointers may become a scalability and performance bottleneck. Hence, reducing contention on any of these pointers may be beneficial. In our prototype implementation, we leave the enqueue operation unmodified, but relax the dequeue operation in a generic way to reduce contention on the head pointer. In particular, we add a field to each queue element for marking elements as already returned but not yet removed from the queue so that elements other than the oldest element can be returned at any time without being immediately removed from the queue [1]. Marked elements are removed later more efficiently when the head pointer is set to a younger element.

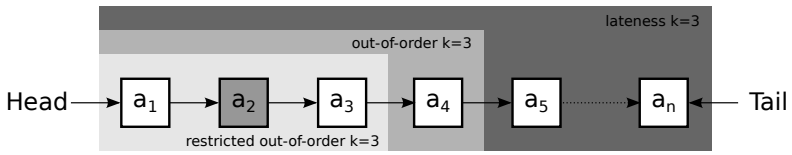


Fig. 1. The ranges of elements which may be returned by a dequeue operation of a k-FIFO queue with restricted out-of-order, out-of-order, and lateness relaxation with $k = 3$. An already returned but not yet removed element (a_2) is marked in grey.

Figure 1 shows the ranges of elements which may be returned by a dequeue operation of a k-FIFO queue with restricted out-of-order relaxation (light grey), out-of-order relaxation (medium grey), and lateness relaxation (dark grey) with $k = 3$. An already returned but not yet removed element (a_2) is marked in grey.

[3] The canonical representative is a matter of choice. Equally justified is using the unique kernel sequence ordered in non-increasing priority, in which case the transitions of a priority queue resemble more the transitions of a stack, highlighting the duality between FIFO queues and stacks.

Listing 1.1. Generic lock-free dequeue operation of a k-FIFO queue

```
1 Object dequeue() {
2   while(true) {
3     Node first = head;
4     Node last = tail;
5     Node next = first.next;
6     if (first == head) {
7       if (first == last) {
8         if (next == null) {
9           return null;
10        }
11        set(tail, next);
12      } else {
13        index = select_unmarked_element(first, k);
14        if (index < 0) {
15          fixup(head);
16        } else {
17          element = get_element(first, index);
18          if (mark(first, index)) {
19            return element;
20          }
21        }
22      }
23    }
24  }
25 }
```

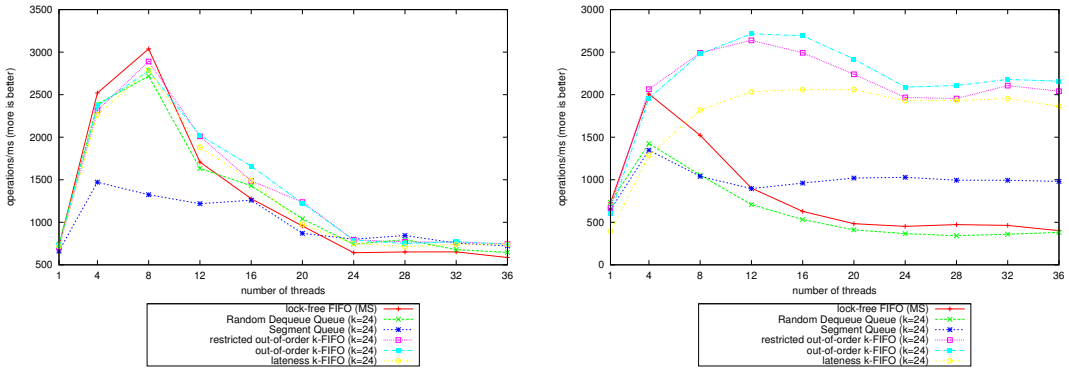
The generic structure of a lock-free dequeue operation of a *k*-FIFO queue is shown in Listing 1.1. If the queue is empty, `null` is returned. If it is not empty, the dequeue operation attempts to select an unmarked element in the queue using the `select_unmarked_element` function, which has to be specified for each concrete relaxation. If no element is selected, e.g. because all elements in the queue have already been returned but not yet removed, the dequeue operation attempts to fix up the head pointer by setting it, using a CAS operation, to the oldest unmarked element, if it exists; otherwise, to `null` to indicate an empty queue. If the fixup of the head pointer fails due to a concurrent head modification by another thread, the dequeue operation retries. If an unmarked element is selected, it is marked using a CAS operation which may fail as well if another thread marks the same element concurrently. In that case, the dequeue operation also retries. Otherwise, the marked element is returned. Note that the head pointer is modified only when all of the *k* oldest elements have already been returned which significantly reduces contention on the head pointer in comparison to the non-relaxed version.

The restricted out-of-order relaxation is implemented in the `select_unmarked_element` function by traversing the linked list of enqueued elements starting from the head pointer and selecting randomly an unmarked element among the *k* oldest elements. The out-of-order relaxation is implemented similarly except that an unmarked element is randomly selected among the *k* oldest unmarked elements independently of the number of marked elements in the queue. The lateness relaxation is implemented using a counter that is incremented atomically whenever an element other than the oldest element is returned. If the counter is less than *k* the `select_unmarked_element` function traverses the linked list of enqueued elements starting from the head pointer and selects randomly an unmarked element in between the queue head and tail. Otherwise, it resets the counter to zero and returns the oldest element. We refer the reader to Appendix B for a proof that these algorithms are instances of out-of-order relaxation.

Experiments. We present performance and scalability data obtained with our previously discussed prototype implementations of the restricted out-of-order, out-of-order, and lateness *k*-FIFO queues. Baseline is the MS FIFO queue. We also include data obtained with our implementations of the so-called Random Dequeue Queue (RQ) and Segment Queue (SQ) [1]. Both RQ and SQ implement restricted out-of-order *k*-FIFO queues. So-called Scal queues [6, 7] also implement relaxed FIFO queues with similar relaxations as ours but are based on distributed FIFO queues and load balancing. Scal queues have already

been shown to provide superior performance and scalability in comparison to many existing regular and relaxed FIFO queue implementations.

All experiments ran on a server machine with four 6-core 2.1GHz AMD Opteron processors (24 cores) and 48GB of memory on Linux 2.6.32. In all experiments the benchmark threads are executed with real-time priorities to minimize system jitter. All algorithms are implemented in C and compiled using gcc 4.3.3 with -O3 optimizations. Allocation and deallocation of queue elements is done thread-locally to minimize cache contention and other scalability issues that may be introduced by the allocator.



(a) Performance of enqueue-dequeue workload

(b) Performance of dequeue-only workload

Fig. 2. Throughput in data structure operations/ms with an increasing number of threads on a 24-core server machine

The data shown in Figure 2(a) was obtained by having each thread alternate between enqueueing and dequeueing an element one million times where the queue is initially empty and $k = 24$. MS shows the best performance for up to eight threads. Beyond that our prototype implementations generally perform better (also than RQ and SQ) but only in an area of negative scalability for all. The data shown in Figure 2(b) was obtained by having each thread dequeue one million elements where the queue is initially filled with t (number of threads) million elements and again $k = 24$. Here our prototype implementations generally show better performance and scalability in comparison to MS, RQ, and SQ. In sum, contention can be reduced using our prototypes but so far only on dequeue-dominated workloads.

7 Related work

The general topic of this paper is part of a recent trend towards scalable but semantically weaker concurrent data structures [11]. We first discuss work related to our framework and then focus on work related to our prototype implementations.

Framework. Our work generalizes previous work on so-called semantical deviation and k-FIFO queues [6] which correspond to restricted out-of-order k-FIFO queues here. So-called weak k-FIFO queues [7] correspond to a combination of out-of-order and k-FIFO queues here.

Our work is closely related to relaxing the semantics of concurrent data structures through quasi-linearizability [1]. Just like quasi-linearizability, we provide quantitative relaxations of concurrent data structures. Unlike quasi-linearizability which uses syntactic distances, our relaxations are based on semantical distances from a sequence to the sequential specification. In our opinion, semantical distances are better suited than syntactic ones, they lead to intuitively expected relaxations. We briefly present

the quasi-linearizability approach here and identify three main differences between our work and quasi-linearizability.

We call two sequences $\mathbf{x}$, $\mathbf{x}'$, both of length n, permutation equivalent, written $\mathbf{x} \sim \mathbf{x}'$, if there exists a permutation p on $\{1,\ldots,n\}$ such that for all $1 \leqslant i \leqslant n$, $\mathbf{x}(i) = \mathbf{x}'(p(i))$. We write $\mathbf{x} \sim_p \mathbf{x}'$ to emphasize the permutation witnessing $\mathbf{x} \sim \mathbf{x}'$. In such a case, the permutation distance between $\mathbf{x}$ and $\mathbf{x}'$ is given as $\max\{|i - p(i)| \mid 1 \leqslant i \leqslant n\}$. Let S be a sequential specification over Σ such that $\mathbf{x}' \in S$ and $\mathbf{x} \notin S$. In [1] the cost of obtaining $\mathbf{x}'$ from $\mathbf{x}$ is defined via a collection D of subsets of Σ. For $A \in D$, let k_A denote the permutation distance between $\mathbf{x}|A$ and $\mathbf{x}'|A$. Then, $\mathbf{x}'$ is quasi-linearizable with quasi-linearization factor $Q\colon D \to \mathbb{N}$, if for all $A \in D$, $k_A \leqslant Q(A)$. The main differences to our work are:

1. *Distance of sequence* $\mathbf{x}$ *to specification* S. For a queue, consider the sequence

$$\texttt{enq(1)enq(2)enq(3)deq(1)deq(2)enq(4)deq(4)}$$

whose last transition has a discarding cost of 1, since the last dequeue operation is executed when the queue contains 3 and 4, with the former at the head. In order to capture this, quasi-linearizability employs a scheme where only `enq` operations are allowed to commute. Formally, quasi-linearizability uses $D = \{Enq, Deq\}$ with $Q(Enq) = k$ and $Q(Deq) = 0$, where Enq (resp., Deq) contains all `enq` (resp., `deq`) symbols. However, with this scheme the sequence

$$\texttt{deq(i)}^n\texttt{enq(i)}^n$$

which removes n elements before these elements are enqueued will always be in any k relaxation of the queue because the permutation equivalent sequence $\texttt{enq(i)}^n\texttt{deq(i)}^n$ will give $k_{Enq} = k_{Deq} = 0$, independent of the value n. In [1] the authors argue that there is no quasi-linearization factor for such implementations, i.e., if an implementation generates such a propheting-deque sequence, then it will generate anything. Thus, quasi-linearizability is not suitable for assigning relaxation costs to single sequences. Observe that the latter sequence contains transitions with ∞ discarding cost, which means that prophetic executions are not allowed at all in our relaxation.

2. *Semantic distance.* For a stack, consider the sequence

$$\mathbf{x} = \texttt{push(a)}[\texttt{push(i)pop(i)}]^n\texttt{push(b)}[\texttt{push(j)pop(j)}]^m\texttt{pop(a)}$$

where prior to the last pop operation, the stack contains `a` and `b`, with the latter at the top position. The distance in the out-of-order relaxation induced by maximal path cost and segment (discarding) cost in this case is 1 since the element popped is immediately after the top entry. However, with quasi-linearization factor Q, it is impossible to precisely capture out-of-order penalty for data structures like stacks. The reason is that in order to get a permutation $\mathbf{x}'$ of $\mathbf{x}$ such that $\mathbf{x}'$ is a valid sequence of a stack, either one of `pop(a)` or `push(b)` has to move over m copies of `push` and `pop` operations, or one of `push(a)` or `push(b)` has to move over n copies of `push` and `pop` operations. So either D is empty which allows for any sequence to be in the relaxation or it is always possible to pick the values for n and m such that the penalty is arbitrarily large.

3. *Limits of permutation.* For a queue, consider the sequence

$$\mathbf{x} = \texttt{enq(1)enq(2)size(4)}$$

This represents a relaxation for the size operation which can return values *close enough* to the actual size of the queue. In this example, the return value of the size operation overshoots by 2, which intuitively should also be the cost of the relaxation. However, since quasi-linearizability can only explore permutation equivalent sequences, such a relaxation is not expressible. Observe that the maximum cost with segment transition cost for $\mathbf{x}$ is 2, obtained by considering any state which has a sequence of length 4 in its kernel.

As opposed to relaxing the sequential specification of a concurrent data structure one may also relax the consistency condition, e.g., quiescent consistency [2] instead of linearizability. We should note that linearizable out-of-order relaxation of a queue is stronger than a quiescently consistent queue. For this, consider a concurrent history **c** with two threads t_1 and t_2. **c** starts with the invocation of `push(a)` by t_1, followed by a sequence $\texttt{pop(i)}^n\texttt{push(i)}^n$ all executed by t_2. This sequence is quiescently consistent for queue because the reordering of methods (even those that do not overlap in time) is allowed as long as they are not separated by a quiescent state. On the other hand, any linearization of **c** will have to observe out-of-order pop operations since `push` and `pop` operations do not overlap. A comprehensive overview of variants of weaker and stronger consistency conditions than linearizability can be found in [4].

Implementation. Our prototype implementations are related to implementations of relaxed FIFO queues such as the previously mentioned Random Dequeue and Segment Queues [1] as well as Scal queues [6, 7]. In [3] the authors show that implementing deterministic data structure semantics requires expensive synchronization mechanisms which may prohibit scalability in high contention scenarios. We agree with that and show in our experiments that the non-determinism introduced in the sequential specification of our k-FIFO queues may result in reduced contention. In [10] the authors present a work-stealing queue with relaxed semantics where queue elements may be returned multiple times instead of just once. In comparison to other state-of-the-art work-stealing queues with non-relaxed semantics this may provide better performance and scalability. Again the introduced non-determinism pays off.

References

1. Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Proc. Conference on Principles of Distributed Systems (OPODIS)*, pages 395–410. Springer, 2010.
2. J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41:1020–1048, 1994.
3. H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proc. of Principles of Programming Languages (POPL)*, pages 487–498. ACM, 2011.
4. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.
5. M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
6. C. Kirsch, H. Payer, H. Röck, and A. Sokolova. Brief announcement: Scalability versus semantics of concurrent FIFO queues. In *Proc. Symposium on Principles of Distributed Computing (PODC)*. ACM, 2011.
7. C. Kirsch, H. Payer, H. Röck, and A. Sokolova. Performance, scalability, and semantics of concurrent FIFO queues. Technical Report 2011-03, Department of Computer Sciences, University of Salzburg, September 2011.
8. I. Lotan and N. Shavit. Skiplist-based concurrent priority queues. In *Proc. International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 263–568. IEEE, 2000.
9. M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. Symposium on Principles of Distributed Computing (PODC)*, pages 267–275. ACM, 1996.
10. M. Michael, M. Vechev, and V. Saraswat. Idempotent work stealing. In *Proc. Principles and Practice of Parallel Programming (PPoPP)*, pages 45–54. ACM, 2009.
11. N. Shavit. Data structures in the multicore age. *Communications ACM*, 54:76–84, March 2011.
12. R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, April 1986.

A FIFO queue, stack, priority queue relaxations

In this section we fill in the missing details from Section 5.

A.1 FIFO queue

To ease the notation, we write S instead of S_{Q} throughout this section. We first show the following two simple properties.

Lemma A1 *If* $\mathbf{s}$ *is a kernel sequences of a FIFO queue, then* $\mathbf{s} \in \{\mathtt{enq(d)} \mid \mathtt{d} \in D\}^*$.

Proof. Let $\mathbf{s} \in S$. If $\mathbf{s} = \mathbf{u} \cdot \mathtt{head(a)} \cdot \mathbf{v}$, then $\mathbf{s} \equiv_S \mathbf{uv}$ and hence it is not minimal in its class. The same holds if $\mathbf{s} = \mathbf{u} \cdot \mathtt{size(n)} \cdot \mathbf{v}$. Now, if $\mathbf{s} = \mathbf{u} \cdot \mathtt{deq(a)} \cdot \mathbf{v}$, then it must be that $\mathbf{u} = \mathbf{u'} \cdot \mathtt{enq(a)} \cdot \mathbf{u''}$ and $\mathbf{s} \equiv_S \mathbf{u'u''v}$, and again $\mathbf{s}$ is not minimal in its class. In all cases, the arguments that a sequence is S-equivalent to another sequence can be made precise by induction on the length of the future. □

Lemma A2 *Let* $\mathbf{s}, \mathbf{s'} \in \{\mathtt{enq(d)} \mid \mathtt{d} \in D\}^*$. *If* $\mathbf{s} \equiv_S \mathbf{s'}$, *then* $\mathbf{s} = \mathbf{s'}$.

Proof. Let $\mathbf{s}, \mathbf{s'} \in \{\mathtt{enq(d)} \mid \mathtt{d} \in D\}^*$ and $\mathbf{s} \neq \mathbf{s'}$. Then it must be the case that $\mathbf{s} = \mathbf{u} \cdot \mathtt{enq(a)} \cdot \mathbf{v}$, $\mathbf{s'} = \mathbf{uw}$ and $\mathbf{w}$ does not start with $\mathtt{enq(a)}$ for some $\mathbf{u} = \mathtt{enq}(\mathtt{a}_1) \ldots \mathtt{enq}(\mathtt{a}_n)$, $n \geqslant 0$, or vice versa. Now take $\mathbf{x} = \mathtt{deq}(\mathtt{a}_1) \ldots \mathtt{deq}(\mathtt{a}_n)$. We have $\mathbf{sx} \cdot \mathtt{deq(a)} \in S$ but $\mathbf{s'x}\mathtt{deq(a)} \notin S$ showing that $\mathbf{s} \not\equiv_S \mathbf{s'}$. □

As a consequence, we get what we were after.

Corollary A3 *Kernel sequences of a FIFO queue are exactly the elements of* $\{\mathtt{enq(d)} \mid \mathtt{d} \in D\}^*$ *and for every state* q *of a FIFO queue, it holds that* $|\ker(q)| = 1$.

Proof. Let $\mathbf{s} \in \{\mathtt{enq(d)} \mid \mathtt{d} \in D\}^*$ and let $\mathbf{k} \in [\mathbf{s}]_S$ be a kernel sequence. By Lemma A1, $\mathbf{k} \in \{\mathtt{enq(d)} \mid \mathtt{d} \in D\}^*$. By Lemma A2 since $\mathbf{k} \equiv_S \mathbf{s}$ we get that $\mathbf{k} = \mathbf{s}$. Hence, the elements of $\{\mathtt{enq(d)} \mid \mathtt{d} \in D\}^*$ are kernel sequences. Another application of Lemma A2 yields that $|\ker(q)| = 1$ for any state q. □

We next present the proof of Proposition 51.

Proof (of Proposition 51). Let $q = [\mathbf{s}]_S, q' = [\mathbf{s'}]_S$ with $\mathbf{s}, \mathbf{s'}$ kernel sequences. Note that from Definition 41 and from the m-out-of-order-k definitions we have that the following are equivalent: (1) $scost(q, m, q') = 0$; (2) $q \xrightarrow{m} q'$; and (3) $\mathbf{s'}$ is m-out-of-order-0 from $\mathbf{s}$. For costs larger than 0, we treat each method separately. Careful inspection of Definition 41 and the transitions of a FIFO queue shows the following:

1. $m = \mathtt{enq(a)}$. Then $scost(q, m, q') = k > 0$ if and only if condition $(1)(i)$ applies for $|\mathbf{v}| = k > 0$ if and only if $\mathbf{s} = \mathbf{uvw}, \mathbf{s'} = \mathbf{u'vw}$, and $[\mathbf{uw}]_S \xrightarrow{\mathtt{enq(a)}} [\mathbf{u'w}]_S$ for $|\mathbf{v}| = k > 0$ if and only if $\mathbf{w} = \varepsilon, \mathbf{u'} = \mathbf{u} \cdot \mathtt{enq(a)}$, $\mathbf{s} = \mathbf{uv}, \mathbf{s'} = \mathbf{u'v}$ for $|\mathbf{v}| = k > 0$ if and only if $\mathbf{s} = \mathbf{uv}, \mathbf{s'} = \mathbf{u} \cdot \mathtt{enq(a)} \cdot \mathbf{v}$ for $|\mathbf{v}| = k > 0$ if and only if $\mathbf{s'}$ is $\mathtt{enq(a)}$-out-of-order-k from $\mathbf{s}$.
2. $m = \mathtt{deq(a)}$. Then $scost(q, m, q') = k > 0$ if and only if condition $(1)(ii)$ applies for $|\mathbf{v}| = k > 0$ if and only if $\mathbf{s} = \mathbf{uvw}, \mathbf{s'} = \mathbf{uvw'}$, and $[\mathbf{uw}]_S \xrightarrow{\mathtt{deq(a)}} [\mathbf{uw'}]_S$ for $|\mathbf{v}| = k > 0$ if and only if $\mathbf{u} = \varepsilon, \mathbf{w} = \mathtt{enq(a)} \cdot \mathbf{w'}$, $\mathbf{s} = \mathbf{vw}, \mathbf{s'} = \mathbf{vw'}$ for $|\mathbf{v}| = k > 0$ if and only if $\mathbf{s} = \mathbf{v} \cdot \mathtt{enq(a)} \cdot \mathbf{w'}, \mathbf{s'} = \mathbf{vw'}$ for $|\mathbf{v}| = k > 0$ if and only if $\mathbf{s'}$ is $\mathtt{deq(a)}$-out-of-order-k from $\mathbf{s}$.
3. $m = \mathtt{deq(null)}$. Then $scost(q, m, q') = k > 0$ if and only if condition $(1)(i)$ or $(1)(ii)$ applies for $|\mathbf{v}| = k > 0$ if and only if (since $\mathtt{deq(null)}$ is only possible in the empty state) $\mathbf{s} = \mathbf{s'} = \mathbf{v}$ for $|\mathbf{v}| = k > 0$ if and only if $\mathbf{s'}$ is $\mathtt{deq(null)}$-out-of-order-k from $\mathbf{s}$.

4. $m = \texttt{head(a)}$. Then $scost(q,m,q') = k > 0$ if and only if condition $(1)(ii)$ applies for $|\mathbf{v}| = k > 0$ if and only if $\mathbf{s} = \mathbf{uvw}, \mathbf{s'} = \mathbf{uvw}$, and $[\mathbf{uw}]_S \overset{\texttt{head(a)}}{\rightarrow} [\mathbf{uw}]_S$, $[\mathbf{u}]_S \overset{\texttt{head(a)}}{\nrightarrow} [\mathbf{u}]_S$ for $|\mathbf{v}| = k > 0$ if and only if $\mathbf{u} = \varepsilon$, $\mathbf{w} = \texttt{enq(a)} \cdot \mathbf{x}$, $\mathbf{s} = \mathbf{vw}, \mathbf{s'} = \mathbf{s}$ for $|\mathbf{v}| = k > 0$ if and only if $\mathbf{s} = \mathbf{v} \cdot \texttt{enq(a)} \cdot \mathbf{x}, \mathbf{s'} = \mathbf{s}$ for $|\mathbf{v}| = k > 0$ if and only if $\mathbf{s'}$ is $\texttt{head(a)}$-out-of-order-k from $\mathbf{s}$.
5. $m = \texttt{size(n)}$. In this case it is not important to distinguish cost 0 from cost larger than 0. We have $scost(q,m,q') = k$ if and only if $\mathbf{s} = \mathbf{s'}$ and one of (1)(i), (1)(ii), (2)(i), or (2)(ii) holds, if and only if $\mathbf{s} = \mathbf{s'}$ and ($\mathbf{s} = \mathbf{uvw}$, $|\mathbf{uw}| = \texttt{n}$, $|\mathbf{v}| = k$ in case $|\mathbf{s}| \geqslant \texttt{n}$ or $\mathbf{s} = \mathbf{uw}$, $|\mathbf{uvw}| = \texttt{n}$, $|\mathbf{v}| = k$ in case $|\mathbf{s}| \leqslant \texttt{n}$), if and only if $\mathbf{s} = \mathbf{s'}$ and ($|\mathbf{s}| = \texttt{n} + k$ or $|\mathbf{s}| = \texttt{n} - k$), if and only if $\mathbf{s} = \mathbf{s'}$ and $||\mathbf{s}| - n| = k$ if and only if $\mathbf{s}$ is $\texttt{size(n)}$-out-of-order-k from $\mathbf{s'}$.

□

A.2 Stack

The set of methods of a stack (including observer methods), with data in a set D and $D_{\texttt{null}} = D \cup \{\texttt{null}\}$, is

$$\Sigma_S = \{\texttt{push(d)} \mid \texttt{d} \in D\} \cup \{\texttt{pop(d)} \mid \texttt{d} \in D_{\texttt{null}}\} \cup \{\texttt{head(d)} \mid \texttt{d} \in D_{\texttt{null}}\} \cup \{\texttt{size(n)} \mid \texttt{n} \in \mathbb{N}\}.$$

The sequential specification S_S consists of all stack-valid sequences, i.e., sequences in which each pop pops the top of the stack, each push pushes an element at the top, each top observes the top of the stack, and each size returns the correct size of the queue. The sequence $\mathbf{s}_S = \texttt{push(a)pop(a)push(b)}$ is in the sequential specification S_S, whereas the sequence $\mathbf{t}_S = \texttt{push(a)push(b)pop(a)}$ is not.

Similarly as for FIFO queue, we can show the following.

Lemma A4 *Kernel sequences of a stack are exactly the elements of* $\{\texttt{push(d)} \mid \texttt{d} \in D\}^*$ *and for every state q of a stack, it holds that* $|\ker(q)| = 1$.

Therefore, the transitions of $LTS(S_S)$ are fully described by

$$[\mathbf{s}]_{S_S} \overset{\texttt{push(a)}}{\rightarrow} [\mathbf{s} \cdot \texttt{push(a)}]_{S_S}, [\mathbf{s}]_{S_S} \overset{\texttt{pop(a)}}{\rightarrow} [\mathbf{s'}]_{S_S} \text{ if } \mathbf{s} = \mathbf{s'} \cdot \texttt{push(a)}, \text{ and } [\mathbf{s}]_{S_S} \overset{\texttt{pop(null)}}{\rightarrow} [\varepsilon]_{S_S} \text{ if } \mathbf{s} = \varepsilon.$$

and for observer methods by

$$[\mathbf{s}]_{S_S} \overset{\texttt{top(a)}}{\rightarrow} [\mathbf{s}]_{S_S} \text{ if } \mathbf{s} = \mathbf{s'} \cdot \texttt{push(a)}, [\mathbf{s}]_{S_S} \overset{\texttt{top(null)}}{\rightarrow} [\mathbf{s}]_{S_S} \text{ if } \mathbf{s} = \varepsilon, \text{ and } [\mathbf{s}]_{S_S} \overset{\texttt{size(n)}}{\rightarrow} [\mathbf{s}]_{S_S} \text{ if } |\mathbf{s}| = \texttt{n}.$$

Let $\mathbf{s}$ be a kernel sequence. A kernel sequence $\mathbf{s'}$ is

- $\texttt{push(a)}$-out-of-order-k from $\mathbf{s}$ if $\mathbf{s'} = \mathbf{u} \cdot \texttt{push(a)} \cdot \mathbf{v}$ where $\mathbf{s} = \mathbf{uv}$, $\mathbf{v}$ is minimal, and $|\mathbf{v}| = k$;
- $\texttt{pop(a)}$-out-of-order-k from $\mathbf{s}$ if $\mathbf{s'} = \mathbf{uv}$ where $\mathbf{s} = \mathbf{u} \cdot \texttt{push(a)} \cdot \mathbf{v}$, $\mathbf{v}$ is minimal, and $|\mathbf{v}| = k$;
- $\texttt{pop(null)}$-out-of-order-k from ε if $|\mathbf{s'}| = k$;
- $\texttt{top(a)}$-out-of-order-k (from itself) if $\mathbf{s'} = \mathbf{u} \cdot \texttt{push(a)} \cdot \mathbf{v}$, $\mathbf{v}$ is minimal, and $|\mathbf{v}| = k$;
- $\texttt{top(null)}$-out-of-order-k from ε if $|\mathbf{s'}| = k$;
- $\texttt{size(n)}$-out-of-order-k (from itself) if $||\mathbf{s'}| - \texttt{n}| = k$.

Just like for a FIFO queue, by inspecting all cases, we can show the following proposition.

Proposition A5 *Let* $\mathbf{s}$ *and* $\mathbf{s'}$ *be two kernel sequences of a stack. Then* $[\mathbf{s}]_{S_S} \overset{m,k}{\rightarrow} [\mathbf{s'}]_{S_S}$ *in the out-of-order relaxation with segment cost if and only if* $\mathbf{s'}$ *is m-out-of-order-k from* $\mathbf{s}$.

Again, the relaxations can be applied method-wise. We implemented k-relaxed stacks with only push and pop methods, of which only pop is relaxed according to the segment cost (discard cost). The interpretation of the path cost functions from Section 4 and the corresponding relaxations are as follows:

- The maximal cost represents the maximal distance from the top of a popped element, leading to an *out-of-order k-stack*. Hence, in an out-of-order k-stack, each pop pops an element that is at most k away from the top.
- The deq-interval cost represents lateness, i.e., the maximal number of consecutive pops needed to pop the top, leading to a *lateness k-stack*. Hence, in a lateness k-stack at most the k-th consecutive pop pops the top.
- The deq-interval restricted maximal cost represents the maximal size of a "shrinking window" starting from the top from which elements can be popped, leading to a *restricted out-of-order k-stack*. In a restricted out-of-order k-stack, each pop removes an element at most $k-l$ away from the top, where l is the current lateness of the top.

A.3 Priority queue

The set of methods (including observer methods) for a priority queue is

$$\Sigma_{\mathcal{P}} = \{\texttt{ins(n)} \mid \texttt{n} \in \mathbb{N}\} \cup \{\texttt{rem(n)} \mid \texttt{n} \in \mathbb{N}_{\texttt{null}}\} \cup \{\texttt{top(n)} \mid \texttt{n} \in \mathbb{N}_{\texttt{null}}\} \cup \{\texttt{size(n)} \mid \texttt{n} \in \mathbb{N}\}$$

where $\mathbb{N}$ is the data and priority set and $\mathbb{N}_{\texttt{null}} = \mathbb{N} \cup \{\texttt{null}\}$.

We first present two simple properties related to kernel sequences of a priority queue, of which the second one is obvious.

Lemma A6 *Kernel sequences of a priority queue are all sequences in* $\{\texttt{ins(n)} \mid \texttt{n} \in \mathbb{N}\}^*$.

Proof. It is easy to see that for any sequence that contains other methods than ins there is a shorter equivalent sequence. Hence, kernel sequences contain only ins methods. For the opposite direction, assume that for a sequences of ins methods there exists a shorter sequence equivalent to it. Then there is a method in the first one that is not in the second, and we can provide a future (sequence of rem methods) that will distinguish them, contradicting the equivalence assumption. □

Lemma A7 *Let* $\mathbf{s}, \mathbf{s}' \in \{\texttt{ins(n)} \mid \texttt{n} \in \mathbb{N}\}^*$. *Then* $\mathbf{s} \equiv_S \mathbf{s}'$ *if and only if* $\mathbf{s} \sim \mathbf{s}'$, *i.e., two kernel sequences are equivalent if and only if they are permutation equivalent.*

Hence, in a priority queue, there may be more than one kernel sequence in a state. Nevertheless, the order provides a canonical representative of a state, that we define next.

Definition A8 *The canonical representative of a state q of a priority queue is the unique kernel sequence ordered in non-decreasing priority (note: the smaller the number the higher the priority). We call a sequence canonical kernel sequence if it is the canonical representative of some state.*

Let $\mathbf{s}$ be a canonical kernel sequence. The transitions of $LTS(S_{\mathcal{P}})$ are fully described by

$$[\mathbf{s}]_{S_{\mathcal{P}}} \xrightarrow{\texttt{ins(n)}} [\mathbf{s} \cdot \texttt{ins(n)}]_{S_{\mathcal{P}}}, [\mathbf{s}]_{S_{\mathcal{P}}} \xrightarrow{\texttt{rem(n)}} [\mathbf{s}']_{S_{\mathcal{P}}} \text{ if } \mathbf{s} = \texttt{ins(n)} \cdot \mathbf{s}', \text{ and } [\mathbf{s}]_{S_{\mathcal{P}}} \xrightarrow{\texttt{rem(null)}} [\varepsilon]_{S_{\mathcal{P}}} \text{ if } \mathbf{s} = \varepsilon.$$

Let $\mathbf{s}$ be a canonical kernel sequence. A canonical kernel sequence $\mathbf{s}'$ is

- rem(n)-out-of-order-k from $\mathbf{s}$ if $\mathbf{s}' = \mathbf{vu}$ where $\mathbf{s} = \mathbf{v} \cdot \texttt{ins(n)} \cdot \mathbf{u}$, $\mathbf{v}$ is minimal, and $|\mathbf{v}| = k$;
- rem(null)-out-of-order-k from ε if $|\mathbf{s}'| = k$;
- top(n)-out-of-order-k (from itself) if $\mathbf{s}' = \mathbf{v} \cdot \texttt{ins(n)} \cdot \mathbf{u}$, $\mathbf{v}$ is minimal, and $|\mathbf{v}| = k$;
- top(null)-out-of-order-k (from itself) if $|\mathbf{s}'| = k$;
- size(n)-out-of-order-k (from itself) if $||\mathbf{s}'| - \texttt{n}| = k$.

Note that ins(n)-transitions are always in order, such transitions have either zero or infinite segment cost.

Again, by inspecting all cases of Definition 41, similarly as for FIFO queue and stack, we can show the following proposition.

Proposition A9 *Let* $\mathbf{s}$ *and* $\mathbf{s}'$ *be two canonical kernel sequences of a priority queue. Then* $[\mathbf{s}]_{S_{\mathcal{P}}} \xrightarrow{m,k} [\mathbf{s}']_{S_{\mathcal{P}}}$ *in the out-of-order relaxation with segment cost if and only if* $\mathbf{s}'$ *is m-out-of-order-k from* $\mathbf{s}$.

We mention the implemented versions of priority queue out-of-order relaxations. They are k-relaxed priority queues with only ins and rem methods, of which only rem is relaxed according to the segment cost (discard cost). The interpretation of the path cost functions from Section 4 and the corresponding relaxations are as follows:

- The maximal cost represents the maximal number of elements with a higher priority than a removed element, leading to an *out-of-order k-priority queue*. Hence, in an out-of-order k-priority queue, each rem removes an element for which there are at most k elements with higher priority in the queue.
- The rem-interval cost represents lateness, i.e., the maximal number of consecutive removes needed to remove a top-priority element, leading to a *lateness k-priority queue*. Hence, in a lateness k-priority queue at most the k-th consecutive rem removes a top-priority element.
- The rem-interval restricted maximal cost represents the maximal size of a "shrinking window" starting from "the" top priority element from which elements can be dequeued, leading to a *restricted out-of-order k-priority queue*. In a restricted out-of-order k-priority queue, each rem removes an element for which there are at most $k - l$ higher-priority elements, where l is the current lateness of "the" top priority element.

B Prototype Implementations and Relaxations

In this section, we will sketch the proof that each prototype implementation, given in Section 6, is a linearizable out-of-order relaxation.

Lemma B1 *The following are satisfied by the prototype implementations given in Sec. 6:*

- *The restricted out-of-order k-FIFO queue implementation is linearizable with respect to the restricted out-of-order k-FIFO queue.*
- *The out-of-order k-FIFO queue implementation is linearizable with respect to the out-of-order k-FIFO queue.*
- *The lateness k-FIFO queue implementation is linearizable with respect to the lateness k-FIFO queue.*

Proof (Sketch). For all implementations, the commit point of a deq operation that returns null is the read of first.next (line 5) in the last iteration of the main loop. For the out-of-order and the restricted out-of order k-FIFO queue implementations, the commit point of a deq operation that returns a non-null value is the step selecting the node to be eventually removed by this operation (line 13). For the lateness k-FIFO queue implementation, the commit point of a deq operation that returns a non-null value is either the atomic increment of the counter keeping track of the number of consecutive times the head element is not removed when the counter has value less than k, or the removal of the node pointed to by head (the current oldest element), in case counter was found to be equal to k. For an enq operation in all implementations, we use the same commit points as that of the original MS queue. For simplicity, we assume that the nodes that cannot be reached from the head pointer are reclaimed by a garbage collector only when no thread can access them.

In order to show that the linearizability claim holds for the given implementations, we have to show that there is a sequential history in which the concurrent methods are ordered according to their commit

points in the respective relaxations of queue. This is equivalent to requiring that whenever a deq commits, the node it reads the value from must satisfy the out-of-order constraints (cf. Fig. 1). First, observe that if a deq returns null, it means that at its commit point, head and tail were pointing to the same node and the next pointer of this common node was null. This implies that at the commit point, the queue is logically empty. For a deq returning a non-null value, we have the following cases.

- For restricted out-of-order k-FIFO queue implementation, the node selected cannot be more than k nodes (marked and unmarked combined) away from the head. This is because the head pointer[4] moves only towards the end of the list, and if at some point q during the execution, head and some node n reachable from head are l nodes apart, then at all points q' occurring after q, either n is reachable from head and they are $j \leqslant l$ nodes apart, or n is not reachable from head. Once n cannot be reached from head, it stays unreachable.
- For out-of-order k-FIFO queue implementation, the node selected cannot be more than k unmarked nodes away from head. The argument is similar to the restricted out-of-order case.
- For the lateness k-FIFO queue implementation, we have two sub-cases to consider:
  Counter less than k. If the counter was $j < k$ prior to incrementing, then it means that at most j many deq's have committed by removing elements all of which are different from the oldest element. This implies that at the commit point for this deq the lateness requirement will not be violated.
  Counter equal to k. Since we assume that this deq operation returned a non-null value, it must be the value contained in the oldest element (the node pointed to by head). As long as the counter's value is equal to k, only the nodes pointed to by head are allowed to be removed, and their commit points (removal of the node from the list) are in correct temporal order. Since all of such concurrent deq calls can only terminate after either resetting the counter or seeing a value less than k, they cannot contribute to the lateness count of any other element. Once the counter is reset to 0 by any of the concurrent deq's, the node pointed to by head at the time of resetting becomes the *new* oldest element (some node n'), satisfying the property that no node reachable from n' could have been removed between the counter being reset and the update of the head pointer.

□

C Shared Counter

In this section, we will further illustrate the application of our framework. We consider a new data structure, called a shared counter. We will define two relaxations on this data structure and follow the steps of our methodology, by defining cost domains, transition and path cost functions for each relaxation. We begin with the formal definition of a *shared counter*.

Definition C1 *A shared counter S_{SC} is a set of sequences over the alphabet $\Sigma_{SC} = \{\texttt{get\&Inc}(i) \mid i \in \mathbb{N}\}$. The empty sequence ε is in S_{sc} and a sequence $\mathbf{x}$ of length $n > 0$ is in S_{sc} iff $\mathbf{x}(i) = \texttt{get\&Inc}(i)$, for all $1 \leqslant i \leqslant n$.*

Monotonicity Relaxation. First, we remove the requirement that the numbers returned by get&Inc are consecutive, but maintain the requirement that in any sequence a number occurs at most once. For instance,

$$\texttt{get\&Inc(2)get\&Inc(4)get\&Inc(1)get\&Inc(5)}$$

is a sequence that will be allowed by this relaxation. Following our framework, we should define, after assuming that $\mathrm{LTS}(S_{SC})$ is completed to $\mathrm{LTS}_c(S_{SC})$, the transition and path cost functions. We choose the cost domain to be $\mathbb{Z} \cup \{\infty\}$.

[4] Strictly speaking, the next pointer of the sentinel node head points to.

For the transition cost, we define a slightly modified version of the segment cost, called *directional* cost. Note that each state q in $\mathrm{LTS}(S_{SC})$ has a unique sequence in its equivalence class and a unique transition enabled at it. In other words, a state q can do the transition $q \xrightarrow{\texttt{get\&Inc}(i)} q'$ in $\mathrm{LTS}(S_{SC})$ iff $q = \ker(q) = \{\texttt{get\&Inc(1)} \ldots \texttt{get\&Inc}(i-1)\}$ and $q' = \{\texttt{get\&Inc(1)} \ldots \texttt{get\&Inc}(i)\}$. For simplicity, we will identify q with the sequence it contains. The directional cost will measure the cost of updating the current state so that the transition it does is enabled.

Definition C2 *The directional cost of transition $t = (q, m, q')$, written as $cost_{dir}(t)$, is equal to*[5] $|\hat{q}| - |q|$ *if there exists $\hat{q}$ such that $\hat{q} \xrightarrow{m} q'$ is a transition in* $\mathrm{LTS}(S_{SC})$*; otherwise, $cost_{dir}(t) = \infty$.*

Observe that t is allowed in $\mathrm{LTS}(S_{SC})$ if and only if its directional cost is 0.

For the path cost, we define a *separation path cost* function, pcost_{SC}, that returns the longest separation between two consecutively generated numbers in the sequence, provided that no number appears more than once in the sequence. Otherwise, if a number occurs more than once in a sequence, then its path cost is set to ∞.

Definition C3 *Let $\tau = (m_1, k_1) \ldots (m_n, k_n)$ be a quantitative trace of* $\mathrm{qtr}(S_{SC})$*. The separation path cost* $\mathrm{pcost}_{SC} : \mathrm{qtr}(S_{SC}) \to \mathbb{Z} \cup \{\infty\}$ *maps τ to ∞ if there exist $1 \leqslant i < j \leqslant n$ such that $m_i = m_j$; otherwise,* $\mathrm{pcost}_{SC}(\tau) = \max\{|k_i| \mid 1 \leqslant i \leqslant n\}$.

Monotonicity relaxation of the shared counter is achieved when instantiating our framework with the directional cost and the separation path cost.

Uniqueness Relaxation. As another alternative relaxation for shared counter implementation, we now consider ignoring the requirement that the numbers generated in a sequence are consecutive but maintain that they remain in a monotonically non-decreasing order. For instance,

$$\texttt{get\&Inc(1)get\&Inc(4)get\&Inc(4)get\&Inc(5)}$$

is a sequence that will be allowed by this relaxation. Note that, the two relaxations, uniqueness and monotonicity, are not comparable.

Again, with $LTS_c(S_{SC})$ as before, we set the cost domain to $\mathbb{Z} \cup \{\infty\}$. For transition cost, we define a new function called *iteration cost*.

Definition C4 *Let $t = (q, m, q')$ be a transition in* $\mathrm{LTS}_c(S_{SC})$*. The iteration cost of t, written as $icost(t)$, is given by $|\mathbf{u}| - 1$, where $\mathbf{u} \in \Sigma^*_{SC}$ is the (unique) sequence such that $q \xrightarrow{\mathbf{u}} q'$.*

Intuitively, there are two categories of transitions allowed by the uniqueness relaxation. The first category consists self-loops at each state. The iteration cost for this category is given as -1, since the only sequence that causes self-loop in the original specification $\mathrm{LTS}(S_{SC})$ is the empty sequence. The second category consists of arbitrarily long invisible iterations for a single transition. The iteration cost for this category is one less than the total number of transitions, which assigns an iteration cost of 0 to a (correct) single iteration. Again, a transition gets cost 0 if and only if it was allowed in the original specification.

If we are interested in counting the maximum leap due to a single transition, we can simply use the path cost function pcost_{max} (see Section 4), which will return the maximum iteration number per transition over any given path. However, imagine that we are interested in how many times the shared counter fails to produce a new number. For that, we can define a new path cost function called *total failure cost* as follows:

[5] Here we use $|q|$ for the length of the unique (kernel) sequence in q.

Definition C5 *Let* $\tau = (m_1, k_1) \dots (m_n, k_n)$ *be a quantitative trace of* $\mathrm{qtr}(S_{SC})$*. The total failure cost,* $\mathrm{pcost}_{fail} : \mathrm{qtr}(S_{SC}) \to \mathbb{Z} \cup \{\infty\}$*, maps* τ *to the number of elements of the form* $(\texttt{get\&Inc}(i), -1)$ *in* τ*,* $\mathrm{pcost}_{fail}(\tau) = |\{i \mid k_i = -1\}|$.

We have thus showed two possible ways to quantitatively relax a shared counter. This example again demonstrates the flexibility of our framework: defining different cost functions (both for transition and path costs) allows to obtain different intuitively desirable relaxations.