Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-015-0373-2

SPIN 2013

Verifying a quantitative relaxation of linearizability via refinement

Kiran Adhikari[1] · **James Street**[1] · **Chao Wang**[1] · **Yang Liu**[2] · **Shaojie Zhang**[3]

Abstract The recent years have seen increasingly widespread use of highly concurrent data structures in both multi-core and distributed computing environments, thereby escalating the priority for verifying their correctness. *Quasi linearizability* is a quantitative variation of the standard *linearizability* correctness condition to allow more implementation freedom for performance optimization. However, ensuring that the implementation satisfies the quantitative aspect of this new correctness condition is often an arduous task. In this paper, we propose the first automated method for formally verifying quasi linearizability of the implementation model of a concurrent data structure with respect to its sequential specification. The method is based on checking a relaxed version of the refinement relation between the implementation model and the specification model through explicit state model checking. Our method can directly handle concurrent systems where each thread or process makes infinitely many method calls. Furthermore, unlike many existing verification methods, it does not require the user to supply annotations of the linearization points. We have implemented the new method in the PAT verification framework. Our experimental evaluation shows that the method is effective in verifying the new quasi linearizability requirement and detecting violations.

Keywords Linearizability · Model checking · Concurrent data structure · Refinement · Quantitative relaxation

✉ Chao Wang
chaowang@vt.edu

[1] Virginia Tech, Blacksburg, Virginia, USA

[2] Nanyang Technological University, Singapore, Singapore

[3] Singapore University of Technology and Design, Singapore, Singapore

1 Introduction

Linearizability [11,12] is a widely used correctness condition for concurrent data structures. A concurrent data structure is linearizable if each of its operations (method calls) appears to take effect instantaneously at some point in time between its invocation and response. Although being linearizable does not necessarily ensure the full-fledged functional correctness, linearizability violations are often clear indicators that the implementation is buggy. In this sense, linearizability serves as a useful correctness requirement for implementing concurrent data structures. However, ensuring linearizability of highly concurrent data structures is a difficult task due to the subtle interactions of concurrent operations and the often astronomically many interleavings. Furthermore, it often makes the implementation less scalable when the number of concurrent threads or processes increases.

Quasi linearizability [1] is a quantitative variation of linearizability [9,10,14,15,21] designed to allow for more flexibility in implementing the concurrent data structures. While preserving the basic intuition of linearizability, quasi linearizability relaxes certain aspects of the semantics of the data structures in order to achieve increased runtime performance. This is motivated by the fact that, in many highly concurrent applications, the underlying data structures do not need to obey their classic semantics. For example, when implementing a queue for the task scheduler in a thread pool, we often do not need to obey the strict first-in-first-out (FIFO) semantics; instead, we may allow the dequeue operation to return any of the first k data items, if such relaxation can help improve the performance of the queue significantly. Here, the only requirement is that such *out-of-order* execution must be bounded within a fixed number of steps. Similarly, when implementing efficient data caching in web applications, we may not need to obey the strict semantics of the underly-

Published online: 18 March 2015

ing data structures since occasionally getting stale data is often acceptable in such applications, as long as the delay of refreshing is bounded. Finally, in distributed systems, the global counter for generating unique identifiers, which is often a performance bottleneck, may be allowed to return out-of-order values occasionally if it helps improve performance.

In this paper, we propose the first fully automated method for verifying this type of relaxed linearizability requirement in the implementation models of concurrent data structures. Practical concurrent data structures that leverage such relaxed notion of linearizability are emerging in recent years [1,9,14,15,21]. Although in the literature, there is a large body of work on formally verifying the standard linearizability, e.g. using techniques such as model checking [5,17,18,31], runtime verification [4,39], and theorem proving [29], these methods cannot be used to directly verify quasi linearizability. Compared to standard linearizability, quasi linearizability is significantly more difficult to verify because in addition to the requirement of effectively covering all possible interleavings of the concurrent events, the verification algorithm also needs to analyze the quantitative aspect of these interleavings.

There are several technical challenges. First, since the number of concurrent operations in each thread or process is allowed to be potentially unbounded, the execution trace may be infinitely long. This precludes the use of existing methods based on systematic testing, such as LineUp [4] and RoundUp [39], because these methods rely on checking permutations of finite-length histories, and they require the user to supply a concrete test program with a fixed number of operations. Second, since the method needs to be fully automated, we do not assume that the user will find and annotate the linearization points of each method. This precludes the use of existing methods that are based on either user guidance (e.g., [29]) or annotated linearization points (e.g., [31]).

In contrast, our new method relies on *model checking* based techniques to formally verify a relaxed version of the refinement relation between the implementation model and the specification model. Given an implementation model, denoted M_{impl}, and a specification model, denoted M_{spec}, we check whether the set of execution traces of M_{impl} is a subset of the execution traces of M_{spec}. Toward this end, we will leverage a classic refinement checking algorithm [17,18] while checking the new quantitative relaxation property.

Consider as an example the problem of verifying that a relaxed queue implementation is quasi linearizable. Our new method will start from the initial state pair $(impl, spec)$, where $spec$ is the initial state of the FIFO queue specification model M_{spec}, and $impl$ is the initial state of the quasi linearizable implementation model M_{impl}. Then, we check whether all subsequent *state transitions* of the implementation model can match some subsequent *state transitions* of the specification model. Recall that program verification in general is an undecidable problem. Therefore, to make sure that our refinement checking problem remains decidable, we assume that the capacity of the data structure in the models are bounded to ensure that the number of states remains finite.

It is worth pointing out that quasi linearizability [1] is a relatively new notion, for which highly concurrent data structures are still being developed. There are methods for making queues and priority queues quasi linearizable, but we are not yet aware of any published method for implementing other quasi linearizable data structures, and the jury is still out on whether quasi-linearizability is applicable to stacks [10].

We have implemented the new method in the PAT verification framework [27]. PAT provides the infrastructure for parsing and analyzing the specification and implementation models written in a process algebra that resembles CSP [13]. It also provides a set of API functions to facilitate the construction of explicit state model checking procedures. Our new method is implemented as a new refinement checking module in PAT and is experimentally compared against the existing refinement checking module in PAT. Our experimental results show that the new method is effective in generating formal proofs when the implementation model is indeed correct, and is effective in detecting violations when the implementation model is buggy.

To sum up, this paper makes the following contributions:

- We formalize the problem of verifying quasi linearizability of concurrent data structures and evaluate the use of an existing refinement checking algorithm for verification, which requires the user to manually construct the relaxed specification model.
- We propose a new refinement checking algorithm to directly verify the quasi linearizability requirement without user intervention, by checking a relaxed version of the refinement relation between the data structure implementation model and the specification model.
- We implement the new method in a software tool based on the PAT verification platform and compare it with the standard refinement checking procedure in PAT. Our experimental results demonstrate the effectiveness of our method on a set of standard and quasi linearizable concurrent data structures including queues, stacks, and priority queues.

The remainder of this paper is organized as follows: We establish notations and review the existing refinement checking algorithm in Sect. 2. We present the overall flow of our method in Sect. 3. We present a manual relaxation approach for verifying quasi linearizability in Sect. 4, which is based on the standard refinement checking algorithm. We present

our fully automated approach in Sect. 5, which is based on a new refinement checking algorithm. We present our experimental results in Sect. 6, review related work in Sect. 7, and finally conclude in Sect. 8.

2 Preliminaries

We define standard and quasi linearizability in this section, and review an existing algorithm for checking the standard refinement relation between two labeled transition systems.

2.1 Linearizability

Linearizability [12] is a safety property of concurrent systems, designed over sequences of actions corresponding to the invocations and responses of the operations on shared objects. We begin by formally defining the underlying sequentially consistent shared memory model of a concurrent system.

Definition 1 (*System models*) A shared memory model $\mathcal{M}$ is a 3-tuple structure $(O, init_O, P)$, where O is a finite set of shared objects, $init_O$ is the initial state of O, and P is a finite set of threads or processes accessing the objects. □

Every shared object has a set of states. Each object supports a set of *operations*, or method calls, which are pairs of invocations and matching responses. These operations are the only means of accessing the state of the object. A shared object is *deterministic* if, given the current state and an invocation of an operation, the next state of the object and the return value of the operation are unique. Otherwise, the shared object is *non-deterministic*.

A *sequential specification* ($spec$) of a deterministic shared object is a function that maps every pair of invocation and object state to a pair of response and new object state. In contrast, a sequential specification of a non-deterministic shared object is a function that maps every pair of invocation and object state to *a set of pairs* of response and new object state. More rigorously, the sequential specification is defined for a *type* of shared objects rather than an individual instance. For simplicity, however, we shall refer to both actual shared objects and their types interchangeably in this paper.

An execution of the sequentially consistent shared memory model $\mathcal{M} = (O, init_O, P)$ is modeled by a history, which is a sequence of operation invocation and response actions that can be performed on O by threads or processes in P. The behavior of $\mathcal{M}$ is defined as the set, H, of all possible histories together with the initial valuation of the objects. A history $\sigma \in H$ induces a partial order $<_\sigma$ on operations such that $op_1 <_\sigma op_2$ if the response of operation op_1 occurs in σ before the invocation of operation op_2. Operations in σ that are not related by $<_\sigma$ are concurrent—they represent, for example, method calls that overlap with each other in time. A history σ is called a *sequential* history if and only if $<_\sigma$ is a strict total order. Otherwise, it is called a *concurrent* history.

Let $\sigma|_i$ be the projection of the history σ on process $p_i \in P$, which is the subsequence of σ consisting of all invocations and responses that are performed by p_i. Let $\sigma|_{o_i}$ be the projection of the history σ on object $o_i \in O$, which is the subsequence of σ consisting of all invocations and responses of operations that are performed on the object o_i.

Every history σ of a sequentially consistent shared memory model $\mathcal{M} = (O, init_O, P)$ must satisfy the following basic properties:

- *Correct interaction* For each process $p_i \in P$, the subsequence $\sigma|_i$ consists of alternating invocations and matching responses, starting with an invocation. This property is used to prevent *pipelining* operations, which means that after invoking an operation, the same process invokes another operation before it sees the response of the first operation.
- *Closedness* Every invocation has a matching response. This property is used to prevent *pending* operations, which are invocations without matching responses. Although this property is not required in the original definition of linearizability [12], adding it will not affect the correctness of our verification result. This is because, by Theorem 2 in [12], for a pending invocation in a linearizable history, we can always extend the history to a complete one and preserve linearizability. We choose to include this property to obviate the discussion for pending invocations.

A sequential history σ is *legal* if it respects the sequential specifications of the objects. More specifically, for each object o_i, there exists a sequence of states $s_0, s_1, s_2, \ldots$ of object o_i such that s_0 is the initial valuation of o_i, and according to the sequential specification of the object, the j-th invocation in $\sigma|_{o_i}$ together with state s_{j-1}, where $j = 1, \ldots$, will generate the j-th response in $\sigma|_{o_i}$ and state s_j. For example, a sequence of read and write operations of an object is *legal* if each read returns the value of the preceding write when such value exists, and otherwise it returns the initial value.

Given a history σ, a *sequential permutation* π of σ is a sequential history in which the set of operations as well as the initial states of the objects are the same as in σ.

Definition 2 (*Linearizability*) Given a model $\mathcal{M} = (O = \{o_1, \ldots, o_k\}, init_O, P = \{p_1, \ldots, p_n\})$. Let H be the set of possible histories of $\mathcal{M}$. We say that $\mathcal{M}$ is linearizable if and only if for any history σ in H, there exists a sequential permutation π of σ such that

1. for every pair of operations op_1 and op_2 in σ, if $op_1 <_\sigma op_2$, then $op_1 <_\pi op_2$; that is, π respects the run-time ordering of operations, and
2. for each object $o_i \in O$ $(1 \leq i \leq k)$, the subsequence $\pi|_{o_i}$ is a legal sequential history; that is, π respects the sequential specification (*spec*) of every object. □

For ease of comprehension, linearizability can be equivalently defined as follows. In every history σ, if we assign increasing time values to all operation invocations and responses, then every operation can be shrunk to a single time point between its invocation time and response time such that the operation appears to be completed instantaneously at this time point. In the literature, this time point is called the *linearization point* [3,19].

2.2 Quasi linearizability

Following the notation introduced by Afek et al. [1], we consider quasi linearizability as a quantitative variation of linearizability [11,12], where a history σ is allowed to be non-linearizable but at the same time, stay within a bounded distance from a linearizable history σ'.

For two histories σ and σ' such that one is the permutation of the other, we define their distance as follows. Let $\sigma = e_1, e_2, e_3, \ldots, e_n$ and $\sigma' = e'_1, e'_2, e'_3, \ldots, e'_n$. Let $\sigma[e]$ and $\sigma'[e]$ be the indices of the event e in σ and σ', respectively. The distance between $\Delta(\sigma, \sigma')$ is defined as follows:

$$\Delta(\sigma, \sigma') = \max_{e \in \sigma} \left\{ |\sigma'[e] - \sigma[e]| \right\}.$$

In other words, the distance between σ and σ' is the maximum distance that an event in σ has to move to arrive at its new position in σ'.

While measuring the distance between two histories, we often care about only a subset of the operations. For example, in a concurrent queue, we may care about the ordering of `enqueue` and `dequeue` operations while ignoring the `size` operation. Furthermore, we may choose to allow `dequeue` operations to be executed *out of order* but keep `enqueue` operations *in order*. In such case, we need a way to specify ordering constraints on a subset of the operations of the shared object.

Let $Domain(o)$ be the set of all operations of a shared object $o \in O$. Let $d \subset Domain(o)$ be a subset of operations of object o. Let $Powerset(Domain(o))$ be the powerset, consisting of all the subsets of $Domain(o)$. Let $D \subset Powerset(Domain(o))$ be a subset of the powerset. In this paper, we may use `enq` and `deq` instead of `enqueue` and `dequeue` to save space.

Definition 3 (*Q*uasi-Linearization Factor) We define the *quasi-linearization factor* as a function $Q_O : D \to \mathbb{N}$, where D is a subset of the powerset of operations and $\mathbb{N}$ is the set of natural numbers.

Example 1 For an implementation of a bounded queue that stores a set X of non-zero data items, we define the domain as $Domain(\texttt{queue}) = \{enq.x, deq.x, deq.0 \mid x \in X\}$, where $enq.x$ denotes the `enqueue` operation for data item x, $deq.x$ denotes the `dequeue` operation for data item x, and $deq.0$ indicates that the queue is empty. Now, we may define two subsets of $Domain(\texttt{queue})$ as follows:

$$d_1 = \{enq.x \mid x \in X\},$$
$$d_2 = \{deq.x \mid x \in X\}.$$

Let $D = \{d_1, d_2\}$, where d_1 is the subset of `enq` events and d_2 is the subset of `deq` events. The distance between the two histories σ and σ', after they are projected to the subset d_1 (or the subset d_2), is defined as $\Delta(\sigma|_{d_1}, \sigma'|_{d_1})$ (or $\Delta(\sigma|_{d_2}, \sigma'|_{d_2})$). If we require that the `enq` calls follow the FIFO order and that the `deq` calls may be out-of-order by at most K steps, the quasi-linearization factor, denoted $Q_{\{\texttt{queue}\}} : D \to \mathbb{N}$, must be defined as follows:

$$Q_{\{\texttt{queue}\}}(d_1) = 0,$$
$$Q_{\{\texttt{queue}\}}(d_2) = K.$$

Definition 4 (*Quasi Linearizability*) Given a model $\mathcal{M} = (O = \{o_1, \ldots, o_k\}, init_O, P = \{p_1, \ldots, p_n\})$. Let H be the set of possible histories of $\mathcal{M}$. We say that $\mathcal{M}$ is quasi-linearizable under the quasi factor $Q_O : D \to \mathbb{N}$ if and only if for any history σ in H, there exists a sequential permutation π of σ such that

- for every pair of operations op_1 and op_2 in σ, if $op_1 <_\sigma op_2$, then $op_1 <_\pi op_2$; that is, π respects the run-time ordering of operations, and
- for each object $o_i \in O$ $(1 \leq i \leq k)$, there exists another sequential permutation π' of π such that
  1. $\pi'|_{o_i}$ is a legal sequential history; that is, π' respects the sequential specification (*spec*) of every object, and
  2. $\Delta((\pi|_{o_i})|_d, (\pi'|_{o_i})|_d) \leq Q_O(d)$ for all $d \in D$.

Notice that this definition subsumes the definition of standard linearizability because if the quasi factor is $Q_O(d) = 0$ for all $d \in D$, then the objects behave as standard linearizable data structures.

Example 2 Consider the three concurrent histories (execution traces) of a relaxed queue as shown in Fig. 1. The first trace is linearizable, because it is a valid permutation of the sequential history where `Enq(Y)` takes effect before `Deq(X)`. The second trace is not linearizable because the first dequeue operation is `Deq(Y)` but the first enqueue operation is `Enq(X)`.

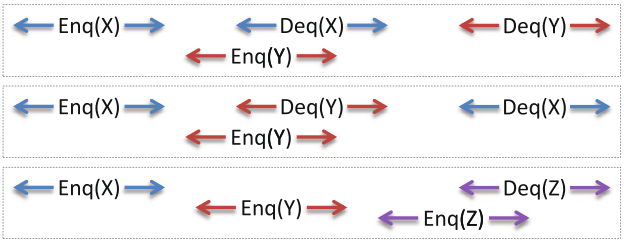

Fig. 1 Three concurrent histories (execution traces) of a relaxed queue: Only the first trace (at the *top*) is linearizable. The second trace is not linearizable, but is 1-quasi linearizable, meaning that the dequeued value may be out of order by at most 1 step. The third trace is neither linearizable nor 1-quasi linearizable

However, note that the second history is not too far away from a linearizable history, since swapping the order of the two dequeue events would make it linearizable again. Therefore, it is considered to be quasi-linearizable with respect to the quasi factor 1. The third trace, in contrast, is neither linearizable nor 1-quasi linearizable, because the dequeue operations are out of order by two steps.

2.3 Linearizability as refinement

Linearizability is defined in terms of the invocations and responses of high-level operations of the shared objects. In a real concurrent program, the high-level operations are implemented by algorithms on concrete shared data structures, e.g., a linked list that implements a shared stack object [28]. Therefore, the execution of high-level operations may have complicated interleaving of low-level actions. Linearizability of a concrete concurrent algorithm requires that, despite low-level interleaving, the history of high-level invocation and response actions still has a sequential permutation that respects both the run-time ordering among operations and the sequential specification of the objects.

For verifying the standard linearizability requirement, an existing method [17,18] can be used to check whether a concurrent algorithm (we refer as *implementation* in this work) refines the high-level linearizable requirement (we refer as *specification* in this work). In this case, the behaviors of the implementation and the specification are modeled as labeled transition systems (LTSs), and the refinement checking is accomplished by using explicit state model checking.

Definition 5 (*Labeled transition system*) A Labeled Transition System (LTS) is a tuple $L = (S, init, Act, \rightarrow)$ where S is a finite set of states; $init \in S$ is an initial state; Act is a finite set of actions; and $\rightarrow \subseteq S \times Act \times S$ is a labeled transition relation.

We write $s \xrightarrow{\alpha} s'$ to denote the state transition $(s, \alpha, s') \in \rightarrow$. The set of enabled actions at s is $enabled(s) = \{\alpha \in Act \mid \exists s' \in S.\ s \xrightarrow{\alpha} s'\}$. A path π of L is a sequence of alternating states and actions $\pi = \langle s_0, \alpha_1, s_1, \alpha_2, \ldots\rangle$ such that $s_0 = init$ and $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all i. If π is finite, then $|\pi|$ denotes the number of transitions in π. In this work, we assume that a path can be infinite, i.e., containing an infinite number of actions. Since the number of states are finite, these infinite paths are paths containing loops. The set of all possible paths for L is written as $paths(L)$.

A transition label can be either a visible action (method invocation or response) or an invisible one (local statement inside a method body). Given an LTS L, the set of visible actions in L is denoted by vis_L and the set of invisible actions is denoted by $invis_L$. A state s' is *reachable* from state s if there exists a path that starts from s and ends with s', denoted by $s \xRightarrow{*} s'$. A τ-transition is a transition labeled with an invisible action. The set of τ-successors is $\tau(s) = \{s' \in S \mid s \xrightarrow{\alpha} s' \wedge \alpha \in invis_L\}$. The set of states reachable from s by performing zero or more τ transitions, denoted as $\tau^*(s)$, can be obtained by repeatedly computing the τ-successors starting from s until a fixed point is reached. We write $s \xrightarrow{\tau*} s'$ iff s' is reachable from s via only τ-transitions, i.e., there exists a path $\langle s_0, \alpha_1, s_1, \alpha_2, \ldots, s_n\rangle$ such that $s_0 = s, s_n = s'$ and $s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \wedge \alpha_{i+1} \in invis_L$ for all $i = 1, \ldots, n-1$.

Given a path π, we can obtain a sequence of visible actions by omitting states and invisible actions. The sequence, denoted as $trace(\pi)$, is a trace of L. The set of all traces of L is written as $traces(L) = \{trace(\pi) \mid \pi \in paths(L)\}$.

Definition 6 (*Refinement*) Let L_1 and L_2 be two LTSs. We say that L_1 refines L_2, written as $L_1 \sqsupseteq_T L_2$ if and only if $traces(L_1) \subseteq traces(L_2)$. □

In an existing work [18], we have shown that if L_{impl} is an implementation LTS and L_{spec} is the LTS of the sequential specification, then L_{impl} is linearizable if and only if $L_{impl} \sqsupseteq_T L_{spec}$.

Algorithm 1 shows the pseudocode of the standard refinement checking procedure in [17,18]. Assume that L_{impl} refines M_{spec}, then for each reachable transition in M_{impl}, denoted as $impl \xrightarrow{e} impl'$, there must exist a reachable transition in L_{spec}, denoted $spec \xrightarrow{e} spec'$. Therefore, the procedure starts with the pair of initial states of the two models, and repeatedly checks whether they have matching successor states. If the answer is no, the check at Lines 6–8 would fail, meaning that L_{impl} is not linearizable. Otherwise, for each pair of immediate successor states, denoted $(impl', spec')$, we add the pair to the *pending* list. The entire procedure continues until either (1) a non-matching transition in L_{impl} is found at Lines 6–8, or (2) all pairs of reachable states are checked, in which case L_{impl} is proved to be linearizable.

In Algorithm 1, the subroutine $next(impl, spec)$ is crucial. It takes the current states of L_{impl} and L_{spec} as input, and returns a set of state pairs of the form $(impl', spec')$. Here,

Algorithm 1 The standard refinement checking algorithm

```
1: Procedure Check-Refinement(impl, spec)
2: checked := ∅
3: pending.push((init_impl, init_spec))
4: while pending ≠ ∅ do
5:    (impl, spec) := pending.pop()
6:    if enabled(impl) ⊈ enabled(spec) then
7:       return false
8:    end if
9:    checked := checked ∪{(impl, spec)}
10:   for all (impl′, spec′) ∈ next(impl, spec) do
11:      if (impl′, spec′) ∉ checked then
12:         pending.push((impl′, spec′))
13:      end if
14:   end for
15: end while
16: return true
```

each pair $(impl', spec')$ is one of the immediate successor state pairs of $(impl, spec)$. They are defined as follows:

1. if $impl \xrightarrow{\tau} impl'$, where τ is an invisible event, then let $spec' = spec$;
2. if $impl \xrightarrow{e} impl'$, where e is a visible event, then follow the transition $spec \xrightarrow{e} spec'$;

In the above presentation, we have assumed without loss of generality that the specification model L_{spec} is deterministic. If the original specification model is non-deterministic, we can always apply standard *subset construction* for finite automaton to make it deterministic.

3 Verifying quasi-linearizability: the overview

Our verification problem is defined as follows: Given an implementation model M_{impl}, a specification model M_{spec}, and a quasi factor Q_O, decide whether M_{impl} is quasi-linearizable with respect to M_{spec} under the quasi factor Q_O. Recall that $Q_o : D \rightarrow \mathbb{N}$ is a function that maps a subset D of the powerset of operations to a natural number $\mathbb{N}$. Whenever the context is clear regarding what D is, we shall use the natural number $QF = Q_o(D)$ to denote the quasi factor.

The straightforward approach for solving the problem is to leverage the standard refinement checking procedure in Algorithm 1. However, since the procedure checks for refinement relation, not quasi refinement relation, the user has to manually construct a relaxed specification model, denoted M'_{spec}, based on the given M_{spec} and quasi factor Q_O. This so-called *manual relaxation* approach is illustrated on the left of Fig. 2. The relaxed specification model M'_{spec} must be able to produce all legal sequential histories that can be produced by M_{spec}, as well as the new sequential histories that are allowed under the relaxed consistency condition given in Definition 4.

This manual relaxation approach, unfortunately, is often difficult to carry out because there is no systematic method or even guideline on how to construct the relaxed specification models. In practice, each M'_{spec} is different, depending on the type of the data structure to be checked. There is often a significant amount of creativity required during the modeling process, e.g., to ensure that the relaxed specification model is both simple and permissive enough.

For example, to verify that a K-segmented queue [1] is quasi-linearizable, we may choose to create a relaxed specification model where the `dequeue` method randomly removes one of the first K data items from the otherwise standard FIFO queue. This new relaxed model M'_{spec} will be more complex than the original specification model M_{spec}, but still simpler than the full fledged implementation model M_{impl}, which requires the use of a segmented linked list.

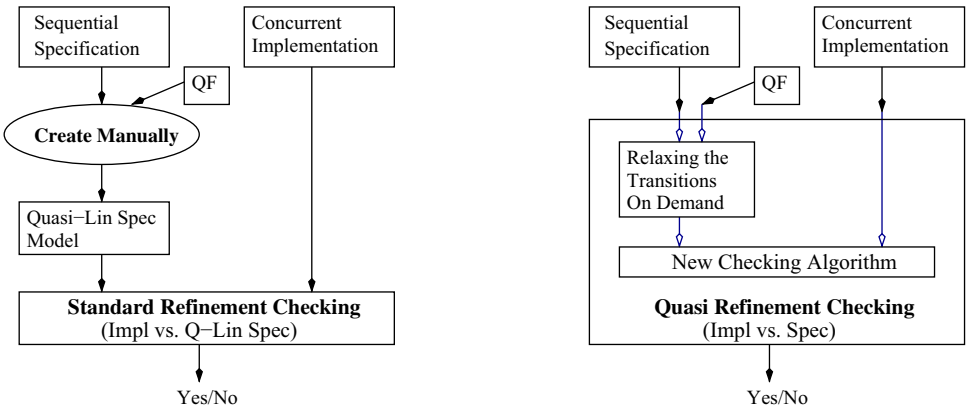

Fig. 2 Two approaches for verifying quasi-linearizability of a concurrent implementation with respect to a sequential specification: in the manual relaxation approach (*left*), the user needs to provide a quasi-linearizable sequential specification; whereas in the automated relaxation approach (*right*), the quasi-linearizable sequential specification is constructed automatically. Here, QF stands for the user given quasi-linearization factor

In contrast, our automated relaxation approach, shown on the right of Fig. 2, relies on a new refinement checking algorithm that can directly check a relaxed refinement relation between M_{impl} and M_{spec}. Therefore, the user does not need to manually construct the relaxed specification model M''_{spec}. Instead, we rely on the algorithm to systematically extend states and transitions of the specification model M_{spec} so that the new states and transitions as required by M'_{spec} are added on the fly. This would lead to the inclusion of a bounded degree of out-of-order execution on the relevant subset of operations as defined by the quasi factor Q_O. A main advantage of our new method is that the procedure is fully automated, thereby avoiding the user intervention, as well as the potential errors that may be introduced during the user's manual relaxation process. Furthermore, by exploring the relaxed transitions on demand, rather than *a priori* as in the manual relaxation approach, we can significantly reduce the number of states that need to be checked.

Since the focus of this paper is on designing a fully automated verification method, we shall briefly illustrate the manual relaxation approach in Sect. 4, and then focus on developing the fully automated approach in subsequent sections.

4 Verifying quasi-linearizability via standard refinement checking

In this section, we will use the standard FIFO queue and k-segmented queue implementations as examples to illustrate the manual relaxation approach. Although we do not intend to promote the manual approach—since it is often labor-intensive and error prone—this section will provide the background information so that we can better illustrate the intuitions behind our fully automated verification method later.

Given the specification model M_{spec} and the quasi factor Q_O, we show how to manually construct the relaxed specification model M'_{spec}. Here, we use the standard FIFO queue and two versions of quasi-linearizable queues as examples. Note that, in general, the construction of the relaxed specification model needs to be tailored case by case for the different types of data structures.

Specification Model M_{spec} The standard FIFO queue with a bounded capacity can be implemented by using a linked list, where the `dequeue` operation removes a data item at one end of the list called the *head* node, and the `enqueue` operation inserts a data item at the other end of the list called the *tail* node. When the queue is full, the `enqueue` operation does not have any impact. When the queue is empty, the `dequeue` returns NULL.

As an example, consider a sequence of four enqueue events `enq(1)`, `enq(2)`, `enq(3)`, `enq(4)`. The subsequent dequeue events would be `deq()=1`, `deq()=2`, `deq()=3`, `deq()=4`, which obey the first-in-first-out semantics. This is illustrated by the first history named `H1-a` in Fig. 4.

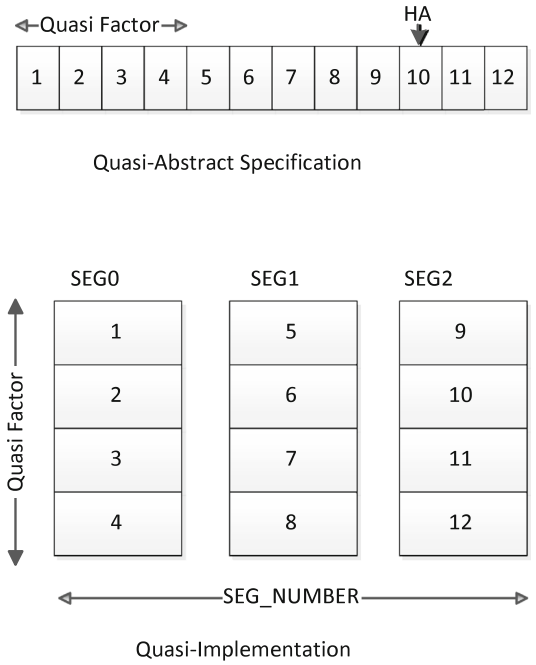

Fig. 3 Two different ways of implementing a 4-quasi queue, where the return value of `deq()` may be out of order by at most 4 data items

H1-a	H1-b	H1-a	H1-b
enq(1)	enq(1)	enq(1)	enq(1)
enq(2)	enq(2)	enq(2)	enq(2)
enq(3)	enq(3)	enq(3)	enq(3)
enq(4)	enq(4)	enq(4)	enq(4)
deq()=1	deq()=1	deq()=2	deq()=2
deq()=2	deq()=2	deq()=1	deq()=1
deq()=3	deq()=4	deq()=3	deq()=4
deq()=4	deq()=3	deq()=4	deq()=3

Fig. 4 The set of valid histories of a *1-quasi linearizable* queue, where the Quasi Factor is set to 1, meaning that the return value of `deq()` is allowed to be out-of-order by at most 1. Specifically, the first `deq()` may return any value from the set {1, 2} and the second `deq()` may return the remaining value. Then, the third `deq()` may return any value from the set {3, 4} and the forth `deq()` may return the remaining value

Implementation Model M_{impl} The bounded quasi-linearizable queue can be implemented using a segmented linked list. This is one of the original algorithms proposed by Afek et al. [1]. A segmented linked list is a linked list where each list node can hold K data items, as opposed to a single data item in the standard linked list. As shown in the lower half of Fig. 3, these K data items form a *segment*, in which the data slots are numbered as 1, 2, . . ., K. In general, the segment size needs to be set to $(QF + 1)$, where QF is the maximum number of out-of-order execution steps.

The example in Fig. 3 has the quasi factor set to 3, meaning that a `dequeue` operation can be executed out of order by at most 3 steps. Consequently, the size of each segment is set to $(3 + 1) = 4$. Since $Q_{\{ttqueue\}}(D_{\texttt{enq}}) = 0$, meaning that the `enqueue` operations should not be reordered, the data items are enqueued regularly in the empty slots of one segment, before the *head* moves to the next segment. But a `dequeue` operation may randomly remove any one of the existing data items from the first segment.

Relaxed specification model M'_{spec} Not all execution traces (histories) of M_{impl} are traces of M_{spec}. In Fig. 4, for instance, histories other than `H1-a` are not linearizable. However, they are all quasi-linearizable under the quasi factor 1. For example, they may be produced by a segmented queue where the segment size is (1+1) = 2. To verify that M_{impl} is indeed quasi-linearizable, we may construct a new specification model M'_{spec}, which includes not only all histories of M_{spec}, but also the histories that are allowed to appear only under the new relaxed consistency condition.

In this example, we choose to construct the new model by slightly modifying the standard FIFO queue. This is illustrated in the upper half of Fig. 3, where the first K data items are grouped into a cluster. Within the same cluster, the `dequeue` operation may remove any of the first k data items based on randomization. Only after the first k data items in the cluster are retrieved, will the `dequeue` operation move to retrieve the next k data items (a new cluster). The external behavior of this model is expected to match that of the segmented queue in M_{impl}; that is, both are *1-quasi-linearizable*.

Checking the refinement relation Once the relaxed specification model M'_{spec} is available, checking whether M_{impl} refines M'_{spec} is straightforward by using Algorithm 1. For the segmented queue implementation [1], for instance, we have manually constructed M'_{spec} and checked the refinement relation in the PAT verification framework. In PAT, both models are written in a process algebra language similar to CSP, called the CSP# [26].

Our experimental results are summarized in Table 1. Column 1 shows the quasi factor of each segmented queue implementation model. Column 2 shows the number of segments in the queue—note that the capacity of the queue is $(QF + 1) \times Seg$. Column 3 shows the refinement checking time in seconds. Column 4 shows the total number of visited states during the refinement checking process. Column 5 shows the total number of state transitions activated during refinement checking. In this section as well as Sect. 6, the allowed data values stored in the data structures are all of *int* type. The experiments are conducted on a computer with an Intel Core-i7, 2.5 GHz processor and 8GB RAM.

The experimental results in Table 1 show an exponential increase in the verification time when we increase the segment size of the queue or the quasi factor. This is due to the fact that the size of the state space grows exponentially. Nevertheless, when the size of the queue is small, we are able to successfully verify quasi-linearizability of the implementation models against the corresponding abstract specifications. However, this method requires the user to manually construct the relaxed specification model M'_{spec}, which is a severe limitation in practice.

Table 1 Experimental results for standard refinement checking. Here, *MOut* means the verification procedure has run out of the 4 GB memory

QF	Seg	Run time (s)	Visited states	Visited transitions
1	1	0.1	423	778
1	2	0.1	2310	4458
1	3	0.1	8002	15,213
1	4	0.4	22,327	41,660
1	5	0.9	55,173	1,01,443
1	6	2.0	126,547	230,259
1	10	55.9	2,488,052	4,421,583
1	15	MOut	–	–
2	1	0.6	26,605	58,281
2	2	12.6	456,397	970,960
2	3	130.7	4,484,213	8,742,485
2	4	MOut	–	–
3	1	8.8	284,484	638,684
3	2	MOut	–	–
4	1	124.4	3,432,702	7,906,856
4	2	MOut	–	–

Manual relaxation is not only labor intensive but also error prone. For example, consider the seemingly simple random dequeued model in the upper half of Fig. 3. A subtle error would be introduced if we do not use the *cluster* to restrict the set of data items that can be removed by the `dequeue` operation. Specifically, assume that the `dequeue` operation always returns one of the first k data items in the current queue. Although it may appear to be correct, such implementation would not have been k-quasi-linearizable, because it is possible for some data item to be over-taken indefinitely. For example, if every time the `dequeue` operation chooses to retrieve *the second data item in the list*, we will have the following `dequeue` sequence:

```
deq()=2, deq()=3, deq()=4, ..., deq()=1,
```

where the retrieval of value 1 is delayed by an arbitrarily long time. This is no longer a *1-quasi-linearizable* queue according to Definition 4.

This example shows that if the user constructs the relaxed specification model M'_{spec} incorrectly, the verification result will be invalid. It motivates us to design a fully automated approach, which can directly check quasi-linearizability of M_{impl} against M_{spec} under the given quasi factor.

5 Verifying quasi-linearizability via a new relaxed refinement checking

In this section, we shall extend the refinement checking procedure in Algorithm 1 to directly check a relaxed refinement relation between M_{impl} and M_{spec}. The idea is to establish a new simulation relation from specification to implementation while allowing relaxation of the specification.

5.1 Linearizability checking via quasi refinement

The new procedure, shown in Algorithm 2, differs from the procedure in Algorithm 1 in the following aspects:

1. We customize the work-list, named *pending*, to make the state exploration follow the breadth-first search (BFS) order. Recall that in Algorithm 1, the exploration may follow either the BFS order or the DFS order, depending on whether *pending* is implemented as a queue or a stack. Here, we insist that it follows the BFS order for ease of implementing the automated relaxation algorithm.
2. We replace the subroutine *enabled*(*spec*) with a new subroutine named *enabled_relaxed*(*spec,QF*). The new subroutine returns not only the events enabled at current *spec* state in M_{spec}, but also the additional events that are allowed under the relaxed consistency condition.
3. We replace the subroutine *next*(*impl,spec*) with a new subroutine named *next_relaxed*(*impl,spec,QF*). The new subroutine will return not only the successor state pairs in the original models, but also the additional successor state pairs that are allowed under the relaxed consistency condition.

Conceptually, the procedure in Algorithm 2 is equivalent to first constructing a relaxed specification model M'_{spec} from (M_{spec}, QF) and then invoking the two newly added subroutines, *enabled(spec)* and *next*(*impl,spec*), on this new model. The main difference from the manual approach is that we are constructing M'_{spec} automatically without the user's intervention. Furthermore, the additional states and edges are added to M'_{spec} during the relaxation process incrementally, on a *need-to* basis.

At the high level, the new procedure performs a BFS exploration starting from the state pair $(impl, spec)$, where *impl* is a state of the implementation model M_{impl} and *spec* is a state of the specification specification model M_{spec}. The initial implementation and specification events are enqueued into *pending*. Each time we go through the while-loop, we first dequeue from *pending* to obtain a state pair, and then check if all events enabled at state *impl* match with some events enabled at state *spec* under the relaxed consistency condition (Line 6). If there is any mismatch, the check fails

Algorithm 2 The new quasi refinement checking algorithm

```
1: Procedure Check-Quasi-Refinement(impl, spec, QF)
2: checked := ∅
3: pending.enqueue((init_impl, init_spec))
4: while pending ≠ ∅ do
5:    (impl, spec) := pending.dequeue()
6:    if enabled(impl) ⊈ enabled_relaxed(spec, QF) then
7:       return false
8:    end if
9:    checked := checked ∪{(impl, spec)}
10:   for all (impl′, spec′) ∈ next_relaxed(impl, spec, QF)
      do
11:      if (impl′, spec′) ∉ checked then
12:         pending.enqueue((impl′, spec′))
13:      end if
14:   end for
15: end while
16: return true
```

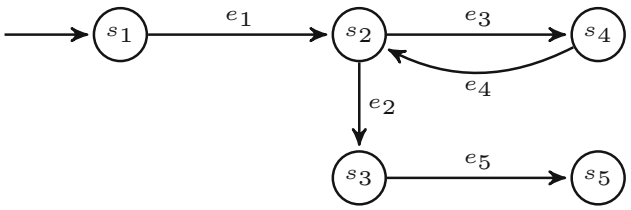

Fig. 5 Specification model before adding the relaxed transitions from s_1

and we can return a counterexample showing how the violation happens. Otherwise, we continue until the work-list *pending* becomes empty. At Lines 10-14, we compute the new successor state pairs by invoking *next_relaxed*, and add these successor state pairs to *pending* if they have not been checked before.

Subroutine enabled_relaxed(*spec,QF*) This subroutine takes the current state *spec* of model M_{spec} and the quasi factor QF as input, and generates all events that can be enabled at state *spec* in the relaxed model M'_{spec}, defined in Sect. 5.2. Consider the graph in Fig. 5 as an example for M_{spec}. Without relaxation, we have $enabled(s_1)$={e_1}. This is equivalent to the result of calling $enabled\_relaxed(s_1, 0)$ when $QF = 0$. However, when the quasi factor $QF = 1$, according to the dotted edges in Fig. 6, $enabled\_relaxed(s_1, 1)$ should return the set $\{e_1, e_2, e_3\}$.

The reason why e_2 and e_3 become enabled is as follows: before relaxation, starting at state s_1, there are two length-3 $(2QF + 1)$ event sequences $\sigma_1 = e_1, e_2, e_5$ and $\sigma_2 = e_1, e_3, e_4$. When $QF = 1$, it means that an event is allowed to be out-of-order by at most 1 step. Therefore, the possible valid permutations of σ_1 are $\pi_1 = e_2, e_1, e_5$ and $\pi_2 = e_1, e_5, e_2$, and the possible valid permutations of σ_2 are $\pi_3 = e_3, e_1, e_4$ and $\pi_4 = e_1, e_4, e_3$. Therefore, at state s_1, events e_2, e_3 can also be executed in the relaxed model. We will discuss the detailed algorithm for generating the valid permutations in Sect. 5.2.

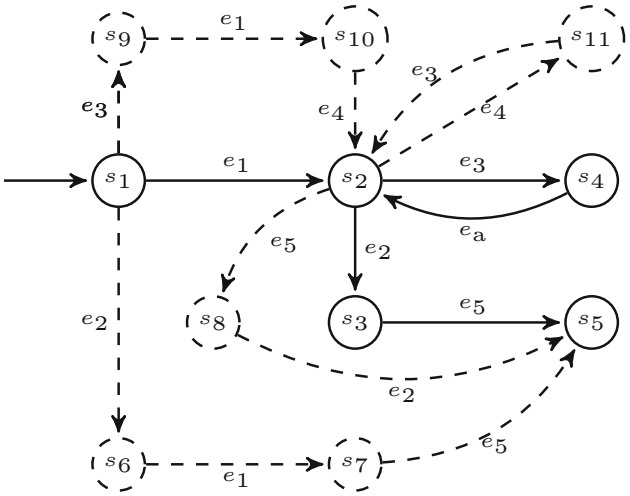

Fig. 6 Specification model after adding the relaxed transitions from s_1. Since the quasi-linearizability factor is set to 1, during relaxation, the original transition sequence $s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_3} s_4 \xrightarrow{e_4} s_2$, for instance, will give rise to the relaxed transition sequence $s_1 \xrightarrow{e_3} s_9 \xrightarrow{e_1} s_{10} \xrightarrow{e_4} s_2$

Subroutine next_relaxed$(impl, spec, QF)$ This subroutine takes the current state $impl$ of M_{impl} and the current state $spec$ of M_{spec} as input, and returns a set of state pairs of the form $(impl', spec')$. The set of state pairs, in general, will be a super set of the set computed by $next(impl, spec)$ in Sect. 2. We define each state pair $(impl', spec')$ as follows:

1. if $impl \xrightarrow{\tau} impl'$, where τ is an invisible event, then let $spec' = spec$;
2. if $impl \xrightarrow{e} impl'$, where e is a visible event, then follow $spec \xrightarrow{e} spec'$ where the visible event e is computed by the subroutine $enabled\_relaxed(spec, QF)$; that is, e is an enabled event at state $spec$ in the relaxed model M'_{spec}.

For example, when $spec = s_1$ in Fig. 5 and the quasi factor is set to 1, it means that the event at state s_1 can be out of order by at most one step. In this case, the procedure *next_relaxed(impl,*s_1, 1) would return not only $(impl', s_2)$, but also $(impl', s_6)$ and $(impl', s_9)$, as indicated by the dotted edges in Fig. 6. The detailed algorithm for generating these relaxed next states in M'_{spec} will be presented in Sect. 5.2.

5.2 Generation of the relaxed specification M'_{spec}

In this subsection, we show how to relax the specification M_{spec} by adding states and transitions that are not allowed by the original specification model but are allowed under the condition of quasi-linearizability, to form the new specification model M'_{spec}. Notice that the relaxation process is carried out incrementally, on a *need-to* basis.

Table 2 Generating the relaxed transition sequences starting from state s_1

BFS	(Frontier)	Event sequences
step 0	$\{s_1\}$	$\langle s_1 \rangle$
step 1	$\{s_2\}$	$\langle s_1 \xrightarrow{e_1} s_2 \rangle$
step 2	$\{s_3, s_4\}$	$\langle s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} s_3 \rangle \langle s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_3} s_4 \rangle$
step 3	$\{s_5, s_2\}$	$\langle s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} s_3 \xrightarrow{e_5} s_5 \rangle \langle s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_3} s_4 \xrightarrow{e_4} s_2 \rangle$

First, for each state $spec$ in M_{spec}, we compute all the event sequences starting at $spec$ with the length $(2QF + 1)$. These event sequences can be computed using a standard BFS based graph traversal algorithm. Figure 5 shows an example for the computation of these event sequences. The specification model M_{spec} has the following set of states $\{s_1, s_2, s_3, s_4, s_5\}$. Suppose that the current state is s_1 (in *step* 0), then the current frontier state set as a result of the BFS traversal is $\{s_1\}$, and the current event sequence is $\langle s_1 \rangle$. The result of each subsequent BFS step is shown in Table 2, also explained as follows:

- In *step* 1, the frontier state set is $\{s_2\}$ and the event sequence becomes $\langle s_1 \xrightarrow{e_1} s_2 \rangle$.
- In *step* 2, the frontier state set is $\{s_3, s_4\}$ and the event sequence is split into two new sequences. One new sequence is $\langle s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} s_3 \rangle$ and the other is $\langle s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_3} s_4 \rangle$.
- The traversal continues until the BFS computation depth finally reaches $(2QF + 1)$.

After completing the $(2QF + 1)$ steps of the BFS traversal starting from the state $spec$, or state s_1 as in the example above, we have to generate all valid permutations of the bounded transition sequences with respect to the quasi factor. This, for example, will transform the original specification model in Fig. 5 into the relaxed specification model in Fig. 6. The dotted states and edges are newly added to reflect the quasi-linearizability relaxation. More specifically, for $QF = 1$, we will reach state s_2 and state s_5 in $(2QF+1) = 3$ steps during the BFS traversal. At *step* 3, there are two existing sequences $\{e_1, e_2, e_5\}$ and $\{e_1, e_3, e_4\}$. For each existing sequence, we compute all possible valid permutation sequences. In this case, the valid permutation sequences are $\{e_2, e_1, e_5\}$,$\{e_1, e_5, e_2\}$ and $\{e_3, e_1, e_6\}$, $\{e_1, e_3, e_6\}$. For each newly generated permutation sequence, we have added new edges and states to the specification model.

Specifically, from an initial state s_1, if we follow the new permutation $\{e_2, e_1, e_5\}$ as shown in Fig. 6, the transition e_2 will lead to the newly formed pseudo state s_6, and the transition e_1 will lead to s_7 from state s_6. From this state, it is then reconnected back to the original state s_5 via transition e_5. Similarly, if we follow the new permutation $\{e_3, e_1, e_4\}$,

the transition e_3 will lead to the newly formed pseudo state s_9, and the transition e_1 will lead to s_{10} from state s_9. From this state, it is then reconnected back to state s_2 via transition e_4. We continue this process of state expansion for all the valid permutation sequences.

Note that this relaxation process needs to be conducted using every state of M_{spec} as the starting point (and for BFS traversal up to $2QF + 1$ steps). This process is also carried out on the fly, each time for a specific starting state, only when it is needed during refinement checking. In particular, the relaxation process shown in Fig. 6 is for the starting state s_1. In general, we need to conduct the same relaxation for all other states in M_{spec}, including s_2, s_3, s_4, and s_5.

Algorithm 3 Expanding the specification model by adding relaxed transition sequences starting from state s_0

```
1: Let s0 be a specification state and QF be the quasi factor
2: Let SEQ = {seq1, seq2, seq3, ··· , seqk} be the set of all possible
   event sequences reachable from s0 in Mspec such that for 1 ≤ i ≤ k,
   each seqi has less than or equal to 2QF + 1 events
3: for all seq in SEQ do
4:   PERMUT_VALID = genValidPermut(seq, QF)
5:   for all perm in PERMUT_VALID do
6:     Let perm = ⟨e1, e2, ··· , en⟩
7:     Let sn be the specification state reached from s0 via seq
8:     if perm is not equal to seq then
9:       for all ei where 1 ≤ i < n do
10:          Create a new state si and a new transition from si−1 to si
             via event ei
11:      end for
12:      Create a new transition from sn−1 to sn via en
13:    end if
14:  end for
15: end for
```

Algorithm 3 shows the pseudocode for expanding the state space of the specification model by starting from state s_0. Let $SEQ = \{seq_1, seq_2, seq_3, ..., seq_k\}$ be the set of sequences that are reachable from the state s_0 in M_{spec} such that each sequence has less than or equal to $(2QF+1)$ events. For each sequence $seq \in SEQ$, we invoke subroutine *genValidPermut(seq,QF)* to compute the set of possible valid permutation paths for that trace (Line 4). Then, a new state is added to the model with a new transition for each event in the permuted sequences, hence allowing the relaxed sequential histories to be admitted to M'_{spec}.

The valid permutations for a given sequence is generated using Algorithm 4, which relies on maintaining a *cost* attribute for each event. It generates all permutations of *seq* while keeping track of the cost of each event, based on which it can avoid generating the invalid permutations, where the cost of any event is outside the trange $[0, 2QF]$.

Initially, for each event e_i in *seq*, where $1 \leq i < n$, the cost is initialized to QF. Then, we compute all possible permutations of the given trace and update the cost of each event

Algorithm 4 $genValidPermut(seq, QF)$

```
1: PERMUT_VALID := ∅
2: Initialize cost associated with each event in seq to QF
3: Generate possible permutations PERMUT_SEQ and update cost
4: for all p in PERMUT_SEQ do
5:   isValid = true
6:   Let p = ⟨e1, e2, ··· , en⟩
7:   for all ei where 1 ≤ i < n do
8:     if ei.cost ≥ 2QF ∨ ei.cost ≤ 0 then
9:       isValid = false
10:      break
11:    end if
12:  end for
13:  if isValid then
14:    PERMUT_VALID = PERMUT_VALID ⋃ p
15:  end if
16: end for
17: return PERMUT_VALID
```

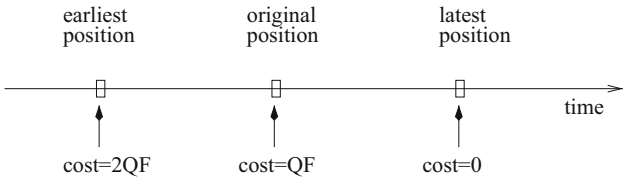

Fig. 7 The positions that an event is allowed to move during relaxation

with respect to its relative position in the new trace. This cost attribute of an event indicates *how many more steps an event is allowed to be postponed* (Fig. 7). Each time an event is postponed, the cost associated with this event is decreased by 1. An event can also be chosen up to QF steps ahead and for each step, the cost is increased by 1. The allowed range for the cost during relaxation is defined as $2QF \leq \text{cost} \leq 0$.

We check the validity of each of these permuted sequences using this cost attribute at Line 8. Only the permutations that pass this check are appended in *PERMUT_VALID*. After the check is completed for all permuted sequences, the subroutine returns. Consider the original event sequence starting at state s_1, $seq = \{e_1, e_2, e_5\}$, as shown in Fig. 5. When $QF = 1$, the cost for each of these events is initialized to 1. Then, we compute all possible permutations by reshuffling the events of the original traces and updating the cost accordingly. In this particular example, there are as 6 possible permutations, which include the two original sequences.

If we consider reordering the sequence $\{e_2, e_1, e_5\}$, for instance, the cost associated with event e_2 would become 2 since e_2 is chosen one step earlier than its original position. For the event e_1, which is postponed for one step, its cost is decreased by 1, which makes the cost associated with e_0 become 0. Event e_3 is not reordered and hence its cost remains unchanged. This sequence is considered to be valid in the relaxed model, because the cost associated with each of the events in this sequence lies within the allowable range.

Table 3 The statistics of the benchmark examples used in our experimental evaluation

Class	Description	Linearizable	Quasi Lin.
Quasi queue (3)	Segmented linked list implementation with quasi factor 2 (seg. size = 3)	No	Yes
Quasi queue (6)	Segmented linked list implementation with quasi factor 2 (seg. size = 6)	No	Yes
Quasi queue (9)	Segmented linked list implementation with quasi factor 2 (seg. size = 9)	No	Yes
Quasi queue (4)	Segmented linked list implementation with quasi factor 3 (seg. size = 4)	No	Yes
Quasi queue (8)	Segmented linked list implementation with quasi factor 3 (seg. size = 8)	No	Yes
Queue buggy1	Segmented queue with a bug (Deq on empty queue may erroneously change current segment)	No	No
Queue buggy2	Segmented queue with a bug (Deq may get value from a wrong segment)	No	No
Lin. queue	A linearizable (hence quasi-linearizable) queue implementation	Yes	Yes
Q. priority Q (3)	Segmented linked list implementation with quasi factor 2 (seg. size = 3)	No	Yes
Q. priority Q (6)	Segmented linked list implementation with quasi factor 2 (seg. size = 6)	No	Yes
Q. priority Q (9)	Segmented linked list implementation with quasi factor 2 (seg. size = 9)	No	Yes
Q. priority Q (4)	Segmented linked list implementation with quasi factor 3 (seg. size = 4)	No	Yes
Priority Q buggy	Segmented priority queue (Deq on empty priority queue may change current segment)	No	No
Lin. stack	A linearizable (hence quasi-linearizable) implementation	Yes	Yes

In contrast, if we consider another permuted sequence $\{e_3, e_1, e_2\}$, where the costs associated with events e_3, e_1, e_2 are $\{3, 0, 0\}$, we know that the sequence is not valid in the relaxed model because the cost of event e_3, which is 3, exceeds the allowable range [2,0].

The complexity of standard refinement checking method as described in Algorithm 1 is $O(N_{spec} \times N_{impl})$, where N_{spec} is the number of states of the specification model and N_{impl} is the number of states of the implementation model. The complexity of the relaxed refinement checking method as described in Algorithm 2 also depends on the quasi factor k, because the specification model may have new states due to the $(k+1)$ permutations of operations starting from each original state. As a result, the number of states in the relaxed specification model increases from N_{spec} to $P(N_{spec}, k+1) = N_{spec} \times (N_{spec} - 1) \times \cdots \times (N_{spec} - k)$. Overall, the complexity is $O(P(N_{spec}, k+1) \times N_{impl})$. When the quasi factor k is significantly smaller than N_{spec}, which is the typical case, the complexity of Algorithm 2 is $O(N_{spec}^{(k+1)} \times N_{impl})$.

6 Experiments

We have implemented our new quasi-linearizability checking method in the PAT verification framework [27]. This new refinement checking algorithm, presented in Sect. 5, can directly check a relaxed version of the refinement relation between the implementation model and the specification model. Since the relaxed refinement relation is a generalization of the standard refinement relation, our new algorithm subsumes the standard refinement checking procedure [17,18] implemented in PAT. In particular, when $QF = 0$, our new procedure degenerates to the standard refinement checking procedure. When $QF > 0$, however, our new procedure has the added capability of checking for the quantitatively relaxed linearizability.

We have evaluated our new algorithm on a set of models of standard and quasi-linearizable concurrent data structures, including linearizable queues, linearizable stacks, quasi queues, and quasi priority queues. For each data structure, we have several variants of the implementation. In addition to the implementations that are known to be linearizable and quasi-linearizable, we also constructed various versions that were thought to be correct initially, but were proved to be buggy subsequently.

The characteristics of all benchmark examples are shown in Table 3. The first two columns list the name of each concurrent data structure and a short description of the implementation. The next two columns show whether the implementation is linearizable and quasi-linearizable.

Table 4 shows the results of our experiments, conducted on a computer with an Intel Core-i7, 2.5 GHz processor and

Table 4 Experimental results for verifying quasi-linearizability (2 threads)

Class	QF	Time (s)	Visited states	Transitions
Quasi queue (3)	2	7.2	126,810	248,122
Quasi queue (6)	2	21.2	237,760	468,461
Quasi queue (9)	2	114.5	1,741,921	3,424,280
Quasi queue (4)	3	131.6	442,558	869,129
Quasi queue (8)	3	1517.1	1,986,924	3,754,489
Queue buggy1	2	0.4	1204	809
Queue buggy2	2	0.1	345	345
Lin. queue	2	5.5	240,583	121,548
Q. priority Q (3)	2	12.2	106,385	195,235
Q. priority Q (6)	2	34.3	472,981	918,530
Q. priority Q (9)	2	198.4	1,478,045	2,905,016
Q. priority Q (4)	3	343.1	1,408,763	2,566,427
Q. priority Q (8)	3	MOut	–	–
Priority Q buggy	2	5.4	894	894
Lin. stack	2	0.2	2690	6896

8 GB RAM running Windows 7. The first column shows the name of each test program. The second column shows the quasi factor. The next three columns show the runtime performance, consisting of the verification time in seconds, the total number of visited states, and the total number of visited transitions.

Our results show that, in general, the number of visited states and the running time increase as the data size increases. For the 3 segmented quasi queue with quasi factor 2, the verification completes in 7.2 s. As the size increases, the time to verify the quasi queue increases. For queue with size 6 and 9, verification is completed in 21.2 s and 114.5 s, respectively. As the quasi factor is increased to 3, the verification time for quasi queue with size 4 and 8 is increased to 131.6 s and 1517.1 s, respectively. This is due to the increase in the size of the state space for the higher quasi factor and segment size. For the priority queues where enqueue and dequeue operations are performed based on the priority, the verification time is higher than the regular quasi queue.

In general, our new method for incremental relaxation of the specification model is significantly more efficient than the manual relaxation approach, because it adds relaxed states and edges to M'_{spec} on a need-to basis, whereas in the manual relaxation approach, the entire relaxed specification model is constructed *a priori*. Specifically, for the 3 segmented quasi queue with quasi factor 2, the verification completes in 7.2 s in the automated approach, in contrast to the 130.7 s used by the manual approach. Figure 8 shows a more detailed comparison of the two approaches on the quasi queues, where the quasi factor is set to 2. The x-axis shows the number of segments of each quasi queue. The y-axis shows the verification time in seconds. When the number of segments of the queue is small,

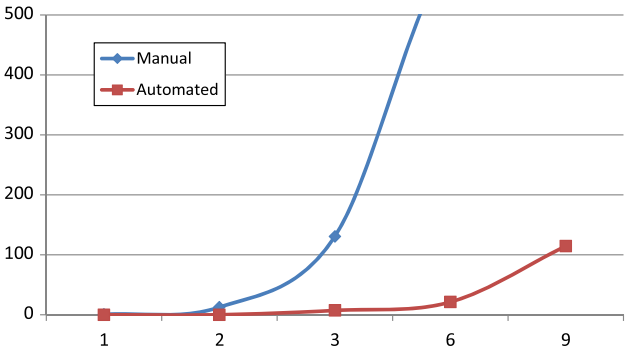

Fig. 8 Comparing the runtime performance of the manual relaxation approach and the automated relaxation approach on the quasi queues

the two approaches have similar runtime performance. When the number of segments becomes larger, however, the automated approach becomes significantly faster than the manual approach, due to its use of on-the-fly relaxation.

It is important to note that since our method relies on constructing the relaxed specification on-the-fly, it often terminates early when the implementation is buggy, thereby generating a counterexample after exploring only part of the state space. This is the reason why in Table 4, the verification time for the buggy queue is shorter than for the correct implementations. It demonstrates one of the strengths our new method in detecting quasi-linearizability violations. Finally, for all test cases shown in Table 4, our method was able to correctly verify the quasi-linearizability requirement or detect the violations.

7 Related work

There is a large body of work in the literature on formally verifying linearizability of concurrent data structures. For example, Liu et al. [18] verify standard linearizability by proving that an implementation model refines a specification model. Vechev et al. [31] use the SPIN model checker to formally verify linearizability in a Promela model with user provided linearization points. Cerný et al. [5] use automated abstractions together with explicit state model checking to verify linearizability in Java programs. There are also works on verifying linearizability by constructing mechanical proofs, which often requires significant manual intervention [29,30]. However, none of these existing methods can directly verify quantitative relaxations of linearizability. Furthermore, it is not immediately clear whether these methods can be extended to achieve this goal.

There are also runtime linearizability checking methods such as LineUp [4], which can directly check the source code implementation of concurrent data structures for violations that can be manifested on finite-length executions. Recently,

we have extended this *runtime* verification method in a new tool, called RoundUp [39], for checking quasi-linearizability violations in C/C++ implementations of concurrent data structures. While both LineUp and RoundUp are effective in detecting implementation bugs in the source/byte code, they require the user to supply concrete test cases. In this sense, they are geared toward runtime bug detection, not static verification of standard/quasi-linearizability.

There are also runtime checking methods for violations of other types of consistency conditions, such as sequential consistency [16], quiescent consistency [2], and eventual consistency [32]. Some of these consistency conditions, in principle, may be used to ensure the correctness of data structures. However, none of these correctness conditions have been as widely used as linearizability for concurrent data structures. Furthermore, unlike quasi-linearizability, these correctness conditions do not have quantitative properties.

Our method is geared toward verifying standard linearizability [11,12] and quasi-linearizability [1] properties of concurrent data structures. In the literature, there are also other types of relaxed concurrent data structures, such as the idempotent work stealing algorithm [20]. Since these algorithms do not have a clear notion of *quasi factor*, it is not immediately clear how our method can be applied to verify them. However, this is an interesting direction to pursue, and we consider it as an item for future work.

Outside the domain of verifying the correctness of concurrent data structures, *serializability* and *atomicity* have been widely used as correctness properties for concurrent programs, especially at the application level. There is a large body of work on both static and dynamic analysis techniques for detecting violations of such properties [6–8,22–25,33–38,40]. These methods differ from ours in that they are checking different type of properties. Although atomicity and serializability are fairly general correctness conditions, they have been applied mostly to shared memory accesses at the load/store instruction level. Linearizability, in contrast, defines the correctness condition for shared objects at the method call level. Furthermore, existing methods for checking atomicity and serializability do not deal with quantitative properties.

8 Conclusions

We have presented a new method for formally verifying quasi-linearizability of the implementation models of concurrent data structures. We have explored two approaches, one of which is based on manually constructing a relaxed specification model, and the other is based on a new algorithm for checking a relaxed refinement relation between the implementation model and the specification model. We have implemented our new method and evaluated it on a set of standard and quasi-linearizable concurrent data structures. Our experiments show that the new method is effective in proving quasi-linearizability and detecting violations. For future work, we plan to incorporate advanced state space reduction techniques such as symmetry reduction and partial order reduction to further improve the performance of our methods.

Acknowledgments This research was supported by the U.S. National Science Foundation under Grant Number CCF-1149454. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Afek, Y., Korland, G., Yanovsky, E.: Quasi-linearizability: relaxed consistency for improved concurrency. In: International Conference on Principles of Distributed Systems, pp. 395–410 (2010)
2. Aspnes, J., Herlihy, M., Shavit, N.: Counting networks. J. ACM. **41**(5), 1020–1048 (1994)
3. Attiya, H., Welch, J.: Distributed computing: fundamentals, simulations, and advanced topics, 2nd edn. John Wiley and Sons Inc., Publication, New Jersey (2004)
4. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 330–340 (2010)
5. Cerný, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent list implementations. In: International Conference on Computer Aided Verification, pp. 465–479 (2010)
6. Farzan, A., Madhusudan, P.: Monitoring atomicity in concurrent programs. In: International Conference on Computer Aided Verification, pp. 52–65 (2008)
7. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 338–349 (2003)
8. Ganai, M.K., Arora, N., Wang, C., Gupta, A., Balakrishnan, G.: BEST: A symbolic testing tool for predicting multi-threaded program failures. In: IEEE/ACM International Conference on Automated Software Engineering, (2011)
9. Haas, A., Lippautz, M., Henzinger, T.A., Payer, H., Sokolova, A., Kirsch, C.M., Sezgin, A.: Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In: Conference Computing Frontiers, p. 17 (2013)
10. Henzinger, T.A., Sezgin, A., Kirsch, C.M., Payer, H., Sokolova, A.:. Quantitative relaxation of concurrent data structures. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 317–328 (2013)
11. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann, USA (2008)
12. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM. Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
13. Hoare, C.A.R.: Communicating sequential processes. Prentice Hall, Englewood Cliffs (1985)
14. Kirsch, C.M. Payer, H.: Incorrect systems: it's not the problem, it's the solution. In: Proceedings of the Design Automation Conference, pp. 913–917 (2012)

15. Kirsch, C.M., Payer, H., Röck, H., Sokolova, A.: Performance, scalability, and semantics of concurrent fifo queues. In: ICA3PP, pp. 273–287 (2012)
16. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE. Trans. Comput. **28**(9), 690–691 (1979)
17. Liu, Y., Chen, W., Liu, Y., Zhang, S., Sun, J., Dong, J.S.: Verifying linearizability via optimized refinement checking. IEEE. Trans. Softw. Eng. (2013)
18. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. In: International Symposium on Formal Methods, pp. 321–337 (2009)
19. Lynch, N.: Distributed algorithms. Morgan Kaufmann, USA (1997)
20. Michael, M.M., Vechev, M.T., Saraswat, V.A.: Idempotent work stealing. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Raleigh, pp. 45–54 (2009)
21. Payer, H., Röck, H., Kirsch, C.M., Sokolova, A.: Scalability versus semantics of concurrent fifo queues. In: ACM Symposium on Principles of Distributed Computing, pp. 331–332 (2011)
22. Said, M., Wang, C., Yang, Z., Sakallah, K.: Generating data race witnesses by an SMT-based analysis. In: NASA Formal Methods, pp. 313–327 (2011)
23. Sinha, A., Malik, S., Wang, C., Gupta, A.: Predicting serializability violations: SMT-based search vs. DPOR-based search. In: Haifa Verification Conference, pp. 95–114 (2011)
24. Sinha, A., Malik, S., Wang, C., Gupta, A.: Predictive analysis for detecting serializability violations through trace segmentation. In: Formal Methods and Models for Codesign, pp. 99–108 (2011)
25. Sinha, N., Wang, C.: Staged concurrent program analysis. In: ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 47–56 (2010)
26. Sun, J., Liu, Y., Dong, J.S., Chen, C.: Integrating specification and programs for system modeling and verification. In: International Symposium on Theoretical Aspects of Software Engineering, pp. 127–135 (2009)
27. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: International Conference on Computer Aided Verification, pp. 709–714 (2009)
28. Treiber, R.K.: Systems programming: coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center (1986)
29. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: International Conference on Verification, Model Checking, and Abstract Interpretation, pp. 335–348 (2009)
30. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 129–136 (2006)
31. Vechev, M.T., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: International SPIN Workshop on Model Checking Software, pp. 261–278 (2009)
32. Vogels, W.: Eventually consistent. Commun. ACM. **52**(1), 40–44 (2009)
33. Wang, C., Ganai, M.: Predicting concurrency failures in generalized traces of x86 executables. In: International Conference on Runtime Verification (2011)
34. Wang, C., Hoang, K.: Precisely deciding control state reachability in concurrent traces with limited observability. In: International Conference on Verification, Model Checking, and Abstract Interpretation, pp. 376–394 (2014)
35. Wang, C., Kundu, S., Ganai, M., Gupta, A.: Symbolic predictive analysis for concurrent programs. In: International Symposium on Formal Methods, pp. 256–272 (2009)
36. Wang, C., Limaye, R., Ganai, M., Gupta, A.: Trace-based symbolic analysis for atomicity violations. In: International Conference on Tools and Algorithms for Construction and Analysis of Systems, pp. 328–342 (2010)
37. Wang, C., Yang, Y., Gupta, A., Gopalakrishnan, G.: Dynamic model checking with property driven pruning to detect race conditions. In: International Symposium on Automated Technology for Verification and Analysis, pp. 126–140 (2008)
38. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multithreaded programs. IEEE. Trans. Softw. Eng. **32**(2), 93–110 (2006)
39. Zhang, L., Chattopadhyay, A., Wang, C.: Round-Up: Runtime checking quasi linearizability of concurrent data structures. In: IEEE/ACM International Conference on Automated Software Engineering, pp. 4–14 (2013)
40. Zhang, L., Wang, C.: Runtime prevention of concurrency related type-state violations in multithreaded applications. In: International Symposium on Software Testing and Analysis, pp. 1–12 (2014)