Verifying a Quantitative Relaxation of Linearizability via Refinement

Kiran Adhikari[1], James Street[1] and Chao Wang[1], Yang Liu[2], and Shao Jie Zhang[3]

[1] Virginia Tech, Blacksburg, Virginia, USA
[2] Nanyang Technological University, Singapore
[3] Singapore University of Technology and Design, Singapore

Abstract. Concurrent data structures have found increasingly widespread use in both multicore and distributed computing environments, thereby escalating the priority for verifying their correctness. *Quasi linearizability* is a relaxation of *linearizability* to allow more implementation freedom for performance optimization. However, ensuring the quantitative aspects of this correctness condition is an arduous task. We propose a new method for formally verifying quasi linearizability of the implementation model of a concurrent data structure. The method is based on checking the refinement relation between the implementation and a specification model via explicit state model checking. It can directly handle concurrent programs where each thread can make infinitely many method calls, and it does not require the user to write annotations for the linearization points. We have implemented and evaluated our method in the PAT model checking toolkit. Our experiments show that the method is effective in verifying quasi linearizability or detecting its violations.

1 Introduction

Linearizability [1, 2] 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 correctness, linearizability violations are clear indicators that the implementation is buggy. In this sense, linearizability serves as a useful correctness condition 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.

Quasi linearizability [3] is a quantitative relaxation of linearizability [4–6] to allow for more flexibility in how the data structures are implemented. While preserving the basic intuition of linearizability, quasi linearizability relaxes the semantics of the data structures to achieve increased runtime performance. For example, when implementing a queue for task schedulers in a thread pool, it is often the case that we do not need the strict first-in-first-out (FIFO) semantics; instead, we may allow the dequeue operations to be overtaken occasionally, if it helps improving the runtime performance. The only requirement is that such out-of-order execution should be bounded by a fixed number of steps. Similarly, when implementing data caching in web applications, we may not need the strict semantics of standard data structures, since occasionally getting stale data is acceptable as long as the delay is bounded. In distributed systems, the counter for generating unique identifiers may also be allowed to return out-of-order values occasionally.

Despite the advantages of quasi linearizability and its rising popularity (e.g., [4–6]), such relaxed consistency property is difficult for testing and validation. Although there is a large body of work on formally verifying linearizability, for example, the methods based on model checking [7–10], runtime verification [11], and mechanical proofs [12, 13], they cannot directly verify quasi linearizability. Quasi linearizability is harder to verify because, in addition to the requirement of covering all possible interleavings of concurrent events, one needs to accurately analyze the quantitative aspects of these interleavings.

In this paper, we propose the first automated method for formally verifying *quasi* linearizability in the implementation models of concurrent data structures. There are several technical challenges. First, since the number of concurrent operations in each thread is unbounded, the execution trace may be infinitely long. This precludes the use of existing methods such as LineUp [11] because they are based on checking permutations of finite histories. 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., [12, 13]) or annotated linearization points (e.g., [9]).

To overcome these challenges, we rely on explicit state model checking. That is, given an implementation model M_{impl} and a specification model 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 extend a classic refinement checking algorithm so that it can check for the newly defined *quantitative relaxation* of standard refinement relation. Consider a quasi linearizable queue as an example. Starting from the pair of initial states of a FIFO queue specification model and its quasi linearizable implementation model, we check whether all subsequent *state transitions* of the implementation model can match some subsequent *state transitions* of the specification model. To make sure that the verification problem remains decidable, we bound the capacity of the data structure in the model, to ensure that the number of states of the program is finite.

We have implemented the new method in the PAT model checking toolkit [14]. PAT provides the infrastructure for parsing and analyzing the specification and implementation models written in a process algebra language that resembles CSP [15]. Our new method is implemented as a module in PAT, and is compared against the existing module for checking standard refinement relation. Our experiments show that the new method is effective in detecting subtle violations of quasi linearizability. When the implementation model is indeed correct, our method can also generate the formal proof quickly.

The remainder of this paper is organized as follows. We establish notations and review the existing refinement checking algorithm in Section 2. We present the overall flow of our new method in Section 3. In Section 4, we present a manual approach for verifying quasi linearizability based on the existing refinement checking algorithm. This approach is labor intensive and error prone, therefore motivating use to design a fully automated method. We present our fully automated method in Section 5, based on our new algorithm for checking the relaxed refinement relation. We present our experimental results in Sections 6. We review related work in Section 7 and give our conclusions in Section 8.

2 Preliminaries

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

2.1 Linearizability

Linearizability [1] is a safety property of concurrent systems, over sequences of actions corresponding to the invocations and responses of the operations on shared objects. We begin by formally defining the shared memory model.

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 valuation of O, and P is a finite set of processes accessing the objects.* □

Every shared object has a set of states. Each object supports a set of *operations*, 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*[4] of a deterministic (resp. non-deterministic) shared object is a function that maps every pair of invocation and object state to a pair (resp. a set of pairs) of response and a new object state. response and a new object state).

An execution of the shared memory model $\mathcal{M} = (O, init_O, P)$ is modeled by a history, which is a sequence of operation invocations and response actions that can be performed on O by processes in P. The behavior of $\mathcal{M}$ is defined as the set, H, of all possible histories together. A history $\sigma \in H$ induces an irreflexive 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. A history σ is $sequential$ iff $<_\sigma$ is a strict total order.

Let $\sigma|_i$ be the projection of σ on process p_i, which is the subsequence of σ consisting of all invocations and responses that are performed by p_i in P. Let $\sigma|_{o_i}$ be the projection of 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 object o_i. Every history σ of a shared memory model $\mathcal{M} = (O, init_O, P)$ must satisfy the following basic properties:

– **Correct interaction:** For each process $p_i \in P$, $\sigma|_i$ consists of alternating invocations and matching responses, starting with an invocation. This property prevents *pipelining*[5] operations.
– **Closedness[6]:** Every invocation has a matching response. This property prevents *pending* operations.

[4] More rigorously, the sequential specification is for a *type* of shared objects. For simplicity, however, we refer to both actual shared objects and their types interchangeably in this paper.
[5] Pipelining operations mean that after invoking an operation, a process invokes another (same or different) operation before the response of the first operation.
[6] This property is not required in the original definition of linearizability in [1]. However adding it will not affect the correctness of our result because by Theorem 2 in [1], for a pending invocation in a linearizable history, we can always extend the history to a complete one and preserve linearizability. We 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 for all $j = 1, 2, \ldots$ according to the sequential specification (the function), the j-th invocation in $\sigma|_{o_i}$ together with state s_{j-1} 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 if there is one, 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 behavior of $\mathcal{M}$. $\mathcal{M}$ is linearizable if for any history σ in H, there exists a sequential permutation π of σ such that*

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

Linearizability can be equivalently defined as follows. In every history σ, if we assign increasing time values to all 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 [16, 17]. This time point is called its *linearization point*.

2.2 Quasi Linearizability

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 histories σ and σ', respectively. The distance between the two histories, denoted $\Delta(\sigma, \sigma')$, is defined as follows:

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

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 position in σ'.

While measuring the distance between two histories, we often care about only a subset of method calls. For example, in a concurrent queue, we may care about the ordering of `enqueue` and `dequeue` operations while ignoring calls to `size` operation. In the remaining of this work, we use words `enq` and `deq` for the interests of space. Furthermore, we may allow `deq` operations to be executed out of order, but keep `enq` operations in order. In such case, we need a way to add ordering constraints on a subset of the methods of the shared object.

Let $Domain(o)$ be the set of all operations of a shared object o. Let $d \subset Domain(o)$ be a subset of operations. Let $Powerset(Domain(o))$ be the set of all subsets of $Domain(o)$. Let $D \subset Powerset(Domain(o))$ be a subset of the powerset.

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

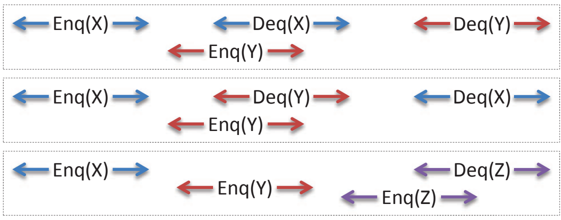

Fig. 1. Execution traces of a queue. Only the first trace (at the top) is linearizable. The second trace is not linearizable, but is 1-quasi linearizable. The third trace is only 2-quasi linearizable.

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

$$d_1 = \{enq.y \mid y \in Y\} \; ,$$
$$d_2 = \{deq.y \mid y \in Y\} \; .$$

Let $D = \{d_1, d_2\}$, where d_1 is the subset of `deq` events and d_2 is the subset of `enq` events. The distance between σ and σ', after being projected to subsets d_1 and d_2, is defined as $\Delta(\sigma|_{d_1}, \sigma'|_{d_2})$. If we require that the `enq` calls follow the FIFO order and the `deq` calls be out-of-order by at most K steps, the quasi-linearization factor $Q_{\{\texttt{queue}\}} : D \to \mathbb{N}$ is 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 behavior of $\mathcal{M}$. $\mathcal{M}$ is quasi linearizable under the quasi factor $Q_O : D \to \mathbb{N}$ if for any history σ in H, there exists a sequential permutation π of σ such that*

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

This definition subsumes the definition for linearizability because, if the quasi factor is $Q_O(d) = 0$ for all $d \in D$, then the objects behaves as a standard linearizable data structure, e.g., a FIFO queue.

Example 2. Consider the concurrent execution of a queue as shown in the Fig. 1. In the first part, it is clear that the execution is linearizable, because it is a valid permutation of the sequential history where `Enq(Y)` takes effect before `Deq(X)`. The second part is not linearizable, because the first dequeue operation is `Deq(Y)` but the first enqueue operation is `Enq(X)`. However, it is interesting to note that the second history is not far from a linearizable history, since swapping the order of the two dequeue events would make it linearizable. Therefore, flexibility is provided in dequeue events to allow them to be reordered. Similarly, for the third part, if the quasi factor is 0 (no out-of-order execution) or 1 (out-of-order by at most 1 step), then the history is not quasi linearizable. However, if the quasi factor is 2 (out-of-order by at most 2 steps), then the third history in Fig.1 is considered as quasi linearizable.

2.3 Linearizability as Refinement

Linearizability is defined in terms of the invocations and responses of high-level operations. 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 [18]. 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 standard (but not quasi) linearizability, an existing method [7, 8] can be used to check whether a real 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.*

For simplicity, we write $s \stackrel{\alpha}{\rightarrow} s'$ to denote $(s, \alpha, s') \in \rightarrow$. The set of enabled actions at s is $enabled(s) = \{\alpha \in Act \mid \exists s' \in S.\ s \stackrel{\alpha}{\rightarrow} s'\}$. A path π of L is a sequence of alternating states and actions, starting and ending with states $\pi = \langle s_0, \alpha_1, s_1, \alpha_2, \cdots \rangle$ such that $s_0 = init$ and $s_i \stackrel{\alpha_{i+1}}{\rightarrow} s_{i+1}$ for all i. If π is finite, then $|\pi|$ denotes the number of transitions in π. A path can also be infinite, i.e., containing infinite number of actions. Since the number of states are finite, 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 or an invisible one. 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 τ-transition is a transition labeled with an invisible action. A state s' is *reachable* from state s if there exists a path that starts from s and ends with s', denoted by $s \stackrel{*}{\Rightarrow} s'$. The set of τ-successors is $\tau(s) = \{s' \in S \mid s \stackrel{\alpha}{\rightarrow} s' \wedge \alpha \in invis_L\}$. 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 $s \stackrel{\tau *}{\rightarrow} s'$ iff s' is reachable from s via only τ-transitions, i.e., there exists a path $\langle s_0, \alpha_1, s_1, \alpha_2, \cdots, s_n \rangle$ such that $s_0 = s$, $s_n = s'$ and $s_i \stackrel{\alpha_{i+1}}{\rightarrow} s_{i+1} \wedge \alpha_{i+1} \in invis_L$ for all i . Given a path π, we can obtain

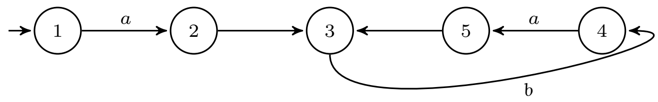

Fig. 2. An LTS example

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 $traces(L)$ = $\{trace(\pi) \mid \pi \in paths(L)\}$.

LTSs can often be shown graphically, e.g., Fig. 2 shows an example LTS, where invisible transition labels are omitted for simplicity. We define the refinement relation between two LTSs, usually called trace refinement, as follows.

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

In [7], we have shown that if L_{impl} is an implementation LTS and L_{spec} is the LTS of the linearizable specification, then L_{impl} is linearizable if and only if $L_{impl} \sqsupseteq_T L_{spec}$.

Algorithm 1 shows the pseudo code of the refinement checking procedure in [7, 8]. 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 as $spec \xrightarrow{e} spec'$. Therefore, the procedure starts with the pair of initial states of the two models, and repeatedly checks whether their 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 $(impl', spec')$, 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 crucially important. 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 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 internal event, then let $spec' = spec$;
2. if $impl \xrightarrow{e} impl'$, where e is a method call event, then $spec \xrightarrow{e} spec'$;

We have assumed, without loss of generality, that the specification model L_{spec} is deterministic. If the original specification model is nondeterministic, we can always apply standard *subset construction* (of DFAs) 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.

Algorithm 1 Standard Refinement Checking

```
Procedure Check-Refinement(impl, spec)
checked := ∅
pending.push((init_impl, init_spec))
while pending ≠ ∅ do
    (impl, spec) := pending.pop()
    if enabled(impl) ⊈ enabled(spec) then
        return false
    end if
    checked := checked ∪{(impl, spec)}
    for all (impl′, spec′) ∈ next(impl, spec) do
        if (impl′, spec′) ∉ checked then
            pending.push((impl′, spec′))
        end if
    end for
end while
return true
```

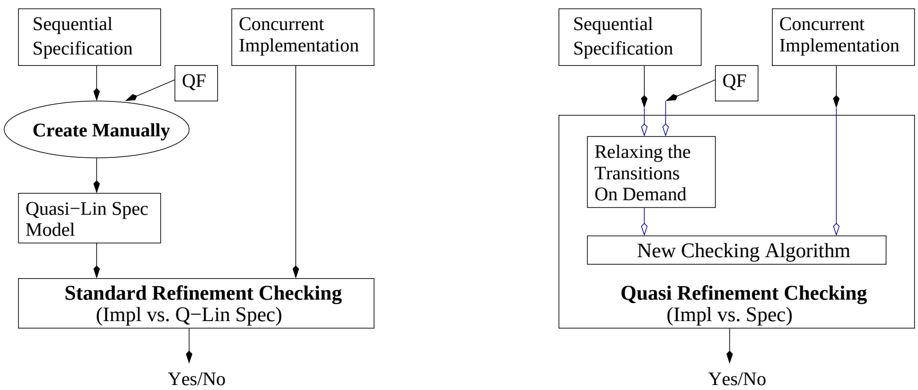

Fig. 3. Verifying quasi linearizability: manual approach (left) and automated approach (right).

The straightforward approach for solving the problem is to leverage the procedure in Algorithm 1. However, since the procedure checks for standard refinement relation, not quasi refinement relation, the user has to manually construct a relaxed specification model, denoted M'_{spec}, based on the given specification model M_{spec} and the quasi factor Q_O. This so-called *manual approach* is illustrated by Fig. 3 (left). The relaxed specification model M'_{spec} must be able to produce all histories that can be produced by M_{spec}, as well as the new histories that are allowed under the relaxed consistency condition in Definition 4.

Unfortunately, there is no systematic method, or general guideline, on constructing such relaxed specification models. Each M'_{spec} may be different depending on the type of data structures to be checked. And there is significant amount of creativity required during the process, to make sure that the new specification model is both simple enough and permissive enough. For example, to verify that a K-segmented queue [3] is quasi linearizable, we can create a relaxed specification model whose `dequeue` method randomly removes one of the first K data items from the otherwise standard FIFO queue. This new model M'_{spec} will be more complex than M_{spec}, but can still be significantly simpler than the full-fledged implementation model M_{impl}, which requires the use of a complex segmented linked list.

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

Our automated approach is shown in Fig. 3 (right). It is based on designing a new refinement checking algorithm that, in contrast to Algorithm 1, can directly check a *relaxed version* of the standard refinement relation between M_{impl} and M_{spec}. Therefore, the user does not need to manually construct the relaxed specification model M''_{spec}. Instead, inside the new refinement checking procedure, we 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 modeling process. Furthermore, by exploring the relaxed transitions on a *need-to* basis, rather than upfront as in the manual approach, we can reduce the number of states that need to be checked.

4 Verifying Quasi Linearizability via Refinement Checking

In this section, we will briefly describe the manual approach and then focus on presenting the automated approach in the subsequent sections. Although we do not intend to promote the manual approach – since it is labor-intensive and error prune – this section will illustrate the intuitions behind our fully automated verification method.

Given the specification model M_{spec} and the quasi factor Q_O, we show how to manually construct the relaxed specification model M'_{spec} in this section. We use the standard FIFO queue and two versions of quasi linearizable queues as examples. The construction 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 `deq` operation removes a data item at one end of the list called the *head* node, and `enq` operation adds a data item at the other end of the list called the *tail* node. When the queue is full, `enq` does not have any impact. When the queue is empty, `deq` 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 FIFO semantics. This is illustrated by the first history `H1-a` in Fig. 5.

In the PAT model checking environment, the specification model M_{spec} is written in a process algebra language, named CSP# [19].

Implementation Model M_{impl}: The bounded quasi linearizable queue can be implemented by using a segmented linked list. This is the original algorithm proposed by Afek *et al.* [3]. 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

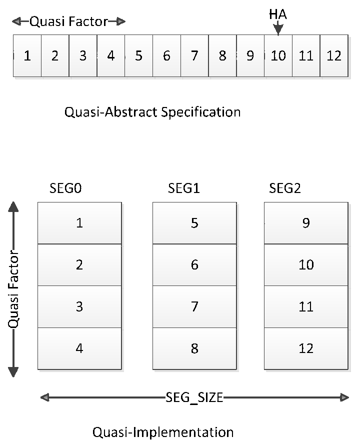

Fig. 4. Implementations of a 4-quasi queue

```
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. 5. Valid histories of a *1-quasi linearizable* queue, meaning that `deq` can be out-of-order by 1. The first `deq` randomly returns a value from the set $\{1, 2\}$ and the second `deq` returns the remaining one. Then the third `deq` randomly returns a value from the set $\{3, 4\}$ and the forth `deq` returns the remaining one.

in Fig. 4 (lower half), 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. 4 has the quasi factor set to 3, meaning that a `deq` 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 $Q_{\{\texttt{queue}\}}(D_{\texttt{enq}}) = 0$, meaning that the `enq` operations cannot be reordered, the data items are enqueued regularly in the empty slots of one segment, before the *head* points to the next segment. But for `deq` operations, we randomly remove one existing data item from the current segment.

Relaxed Specification Model M'_{spec}: Not all execution traces of M_{impl} are traces of M_{spec}. In Fig. 5, histories other than `H1-a` are not linearizable. However, they are all quasi linearizable under the quasi factor 1. They may be produced by a segmented queue where the segment size is (1+1)=2. To verify that M_{impl} is quasi linearizable, we construct a new model M'_{spec}, which includes not only all histories of M_{spec} but also the histories that are allowed only under the relaxed consistency condition. In this example, we choose to construct the new model by slightly modifying the standard FIFO queue. This is illustrated in Fig. 4 (upper half), where the first K data items are grouped into a cluster. Within the cluster, the `deq` operation may remove any of the k data items based on randomization. Only after the first k data items in the cluster are retrieved, will the `deq` move to 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}: both are *1-quasi linearizable*.

Checking Refinement Relation: Once M'_{spec} is available, checking whether M_{impl} refines M'_{spec} is straightforward by using Algorithm 1. For the segmented queue implementation [3], we have manually constructed M'_{spec} and checked the refinement relation in the PAT model checking environment. Our experimental results are summarized in Table 1. Column 1 shows the different quasi factors. Column 2 shows the number of segments – 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

Table 1. Experimental results for standard refinement checking. MOut means memory-out.

Quasi Factor	#. Segment	Verification Time (s)	#. Visited State	#. Transition
1	1	0.1	423	778
1	2	0.1	2310	4458
1	3	0.1	8002	15213
1	4	0.4	22327	41660
1	5	0.9	55173	101443
1	6	2.0	126547	230259
1	10	55.9	2488052	4421583
1	15	MOut	-	-
2	1	0.6	26605	58281
2	2	12.6	456397	970960
2	3	130.7	4484213	8742485
2	4	MOut	-	-
3	1	8.8	284484	638684
3	2	MOut	-	-
4	1	124.4	3432702	7906856
4	2	MOut	-	-

during refinement checking. Column 5 shows the total number of state transitions activated during refinement checking. The experiments are conducted on a computer with an Intel Core-i7, 2.5 GHz processor and 8 GB RAM running Ubuntu 10.04.

The experimental results in Table 1 show an exponential increase in the verification time when we increase the size of the queue or the quasi factor. This is inevitable since the size of the state space grows exponentially. However, this method requires the user to manually construct M'_{spec}, which is a severe limitation because it is often labor intensive and error prone.

For example, consider the seemingly simple random dequeued model in Fig. 4. 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 `deq` operation. Assume that `deq` always returns one of the first k data items in the current queue. Although it appears to be correct, such implementation will not be k-quasi linearizable, because it is possible for some data item to be over-taken indefinitely. For example, if every time `deq` chooses *the second data item in the list*, we will have the following `deq` sequence: `deq.2`, `deq.3`, `deq.4`, . . ., `deq.1`, where the dequeue of value 1 can be delayed by an arbitrarily long time. This is no longer a *1-quasi linearizable* queue. In other words, if the user construct M'_{spec} incorrectly, the verification result becomes invalid.

Therefore, we need to design a fully automated method to directly verify quasi linearizability of M_{impl} against M_{spec} under the given quasi factor QF.

5 New Algorithm for Checking the Quasi Refinement Relation

We shall start with the standard refinement checking procedure in Algorithm 1 and extend it to directly check a relaxed version of the refinement relation between M_{impl} and M_{spec} under the given quasi factor. The idea is to establish the simulation relationship from specification to implementation while allowing relaxation of the specification.

5.1 Linearizability Checking via Quasi Refinement

The new procedure, shown in Algorithm 2, is different from Algorithm 1 as follows:

Algorithm 2 Quasi Refinement Checking

```
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
```

1. We customize *pending* to make the state exploration follow a breadth-first search (BFS). In Algorithm 1, it can be either BFS or DFS based on whether *pending* is a queue or stack.
2. We replace *enabled(spec)* with *enabled_relaxed(spec,QF)*. It will return not only the events enabled at current $spec$ state in M_{spec}, but also the additional events allowed under the relaxed consistency condition.
3. We replace *next(impl,spec)* with *next_relaxed(impl,spec,QF)*. It will return not only the successor state pairs in the original models, but also the additional pairs allowed under the relaxed consistency condition.

Conceptually, it is equivalent to first constructing a relaxed specification model M'_{spec} from (M_{spec}, QF) and then computing the *enabled(spec)* and *next(impl,spec)* on this new model. However, in this case, we are constructing M'_{spec} automatically, without the user's intervention. Furthermore, the additional states and edges that need to be added to M'_{spec} are processed incrementally, on a *need-to* basis.

At the high level, the new procedure performs a BFS exploration for the state pair $(impl, spec)$, where *impl* is the state of implementation and *spec* is a state of specification. The initial implementation and specification events are enqueued into *pending* and each time we go through the while-loop, we dequeue from *pending* a state pair, and 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 and we can return a counterexample showing how the violation happens. Otherwise, we continue until *pending* is empty. Lines 10-14 explore the new successor state pairs, by invoking *next_relaxed* and add to *pending* if they have not been checked.

Subroutine enabled_relaxed(spec,QF): It takes the current state $spec$ of model M_{spec}, along with the quasi factor QF, and generates all events that are enabled at state $spec$.

Consider the graph in Fig. 6 as M_{spec}. Without relaxation, $enabled(s_1)$={e_1}. This is equivalent to $enabled\_relaxed(s_1, 0)$. However, when $QF = 1$, according to the dotted edges in Fig. 7, the set $enabled\_relaxed(s_1, 1)$={e_1, e_2, e_3}.

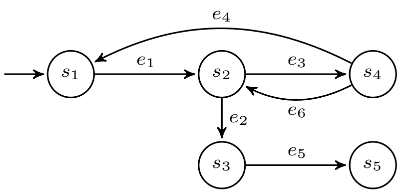

Fig. 6. Specification model before the addition of relaxed transitions for state s_1.

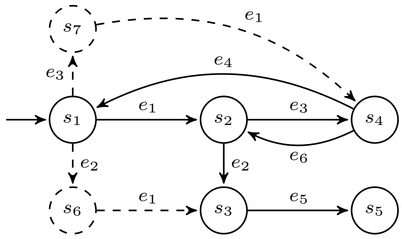

Fig. 7. Specification model after adding relaxed edges for state s_1 and quasi factor 1.

The reason why e_2 and e_3 become enabled is as follows: before relaxation, starting at state s_1, there are two length-2 event sequences $\sigma_1 = e_1, e_2$ and $\sigma_2 = e_1, e_3$. When $QF = 1$, it means an event can be out-of-order by at most 1 step. Therefore, the permutation of σ_1 is $\pi_1 = e_2, e_1$, and the permutation of σ_2 is $\pi_2 = e_3, e_1$. In other words, at state s_1, events e_2, e_3 can also be executed.

Subroutine next_relaxed$(impl, spec, QF)$*:* It 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 $(impl', spec')$. Similar to the definition of $next(impl, spec)$ in Section 2, we define each pair $(impl', spec')$ as follows:

1. if $impl \xrightarrow{\tau} impl'$, where τ is an internal event, then let $spec' = spec$;
2. if $impl \xrightarrow{e} impl'$, where e is a method call event, then $spec \xrightarrow{e} spec'$ where event $e \in enabled\_relaxed(spec, QF)$ is enabled at $spec$ after relaxation.

For example, when $spec = s_1$ in Fig. 6, and the quasi factor is set to 1 – meaning that the event at state s_1 can be out-of-order by at most one step – the procedure *next_relaxed(impl,*s_1*, 1)* would return not only $(impl', s_2)$, but also $(impl', s_6)$ and $(impl', s_7)$, as indicated by the dotted edges in Fig. 7. The detailed algorithm for generation of the relaxed next states in specification is described in Section 5.2.

5.2 Generation of Relaxed Specification

In this subsection, we show how to relax the specification M_{spec} by adding new states and transitions – those that are allowed under the condition of quasi linearizability – to form a new specification model. Notice that we accomplish this automatically, and incrementally, on a *need-to* basis.

For each state $spec$ in M_{spec}, we compute all the event sequences starting at $spec$ with the length ($QF + 1$). These event sequences can be computed by using a simple graph traversal algorithm, e.g. a breadth first search.

Fig. 6 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 is $\{s_1\}$, and the current event sequence is $\langle s_1 \rangle$. The results of each BFS step is shown in Table 2. 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 sequences. One is

Table 2. Specification Sequence Generation at State s_1

BFS Steps	(Frontier)	EventSequences
$step$ 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	$\{s_5, s_2, s_1\}$	$\langle s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} s_3, s_5 \rangle \langle s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_3} s_4 \xrightarrow{e_6} s_2 \rangle \langle s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_3} s_4 \xrightarrow{e_4} s_1 \rangle$

$\langle s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} s_3 \rangle$ and the other is $\langle s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_3} s_4 \rangle$. The traversal continues until the BFS depth reaches ($QF + 1$).

After completing the ($QF + 1$) steps of BFS starting at state $spec$, as described above, we are able to evaluate the following subroutines:

- $enabled\_relaxed(spec, QF)$,
- $next\_relaxed(impl, spec, QF)$.

Consider the quasi factor $QF = 0$ in Fig. 6. In this case, only event e_1 is enabled at s_1. For $QF = 1$, however, events e_2, e_3 are also enabled, as shown by the results at $step$ 2 in Table 2. For $QF = 2$, events e_4, e_5, e_5 are also enabled, as shown by the results at $step$ 3 in the table.

We transform the original specification model in Fig. 6 to the relaxed specification model in Fig. 7 for $QF = 1$. The dotted states and edges are newly added to reflect the relaxation. More specifically, for $QF = 1$, we will reach $(QF + 1) = 2$ steps during the BFS. At $step$ 2, there are two existing sequences $\{e_1, e_2\}$ and $\{e_1, e_3\}$. For each existing sequence, we compute all possible permutation sequences. In this case, the permutation sequences are $\{e_2, e_1\}$ and $\{e_3, e_1\}$. For each newly generated permutation sequence, we add new edges and states to the specification model. From an initial state s_1, if we follow the new permutation $\{e_2, e_1\}$, as shown in Fig. 7, the transition e_2 will lead to newly formed pseudo state s_6 and from this state it is reconnected back to the original state s_3 via transition e_1. Similarly, if we follow the new permutation $\{e_3, e_1\}$, the transition e_3 will lead to newly formed pseudo state s_7, and from this state it is reconnected back to state s_4 via transition e_1.

This relaxation process needs to be conducted by using every existing state of M_{spec} as the starting point (for BFS up to $QF + 1$ steps) and then adding the new states and edges. In our new algorithm, this process is conducted on the fly.

The pseudo code of this relaxation process is shown in Algorithm 3 explains the high level pseudo-code for expanding the state space for the current specification state under the check. Let $SEQ = \{seq_1, seq_2, seq_3, ..., seq_k\}$ be the sequences which are reachable from the state s_0 in M_{spec} for a given quasi factor QF. Each sequence $seq \in SEQ$ is permuted to get all the possible paths for that trace. A new state is formed with a new transition for each event in the permuted sequences, thereby allowing the relaxed refinement checking of the implementation trace.

6 Experiments

We have implemented and evaluated the quasi linearizability checking method in the PAT verification framework [14]. Our new algorithm can directly check a relaxed ver-

Algorithm 3 Pseudo-code for Expanding Specification Under Check

Let s_0 be a specification state and *QF* be the quasi factor
Let $SEQ = \{seq_1, seq_2, seq_3, \cdots, seq_k\}$ be the set of all possible event sequences reachable from s_0 in M_{spec} such that for $1 \leq i \leq k$, the length of seq_i is less than or equal to $QF + 1$
for all *seq* in *SEQ* **do**
    Let *PERMUT_SEQ* be the set of permutations of *seq*
    for all *perm* in *PERMUT_SEQ* **do**
        Let *perm* = $\langle e_1, e_2, \cdots, e_n \rangle$
        Let s_n be the specification state reached from s_0 via *seq*
        if *perm* is not equal to *seq* **then**
            for all e_i where $1 \leq i < n$ **do**
                Create a new state s_i and a new transition from s_{i-1} to s_i via event e_i
            end for
            Create a new transition from s_{n-1} to s_n via e_n
        end if
    end for
end for

Table 3. Statistics of Benchmark Examples

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

sion of the refinement relation. This new algorithm subsumes the standard refinement checking procedure that has already been implemented in PAT. In particular, when $QF = 0$, our new procedure degenerates to the standard refinement checking procedure. When $QF > 0$, our new procedure has the added capability of checking for the quantitatively relaxed refinement relation. Our algorithm can directly handle the implementation model M_{impl}, the standard (not quasi) specification model M_{spec}, and the quasi factor QF, thereby completely avoiding the user's intervention.

We have evaluated our new algorithm on a set of models of standard and quasi linearizable concurrent data structures [3, 5, 6, 4], including queues, stacks, quasi queues, quasi stacks, and quasi priority queues. For each data structure, there can be several variants, each of which has a slightly different implementation. In addition to the implementations that are known to be linearizable and quasi linearizable, we also have versions which initially were thought to be correct, but were subsequently proved to be buggy by our verification tool. The characteristics of all benchmark examples are shown in Table 3. The first two columns list the name of the concurrent data structures

Table 4. Results for Checking Quasi Linearizability with 2 threads and $QF = 2$

Class	Verification Time (s)	Number of Visited States	Number of Visited Transitions
Quasi Queue(3)	7.2	126,810	248,122
Quasi Queue(6)	21.2	237,760	468,461
Quasi Queue(9)	114.5	1,741,921	3,424,280
Queue buggy1	0.4	1,204	809
Queue buggy2	0.1	345	345
Lin. Queue	5.5	240,583	121,548
Q. Priority Queue(3)	12.2	106,385	195,235
Q. Priority Queue(6)	34.3	472,981	918,530
Q. Priority Queue(9)	198.4	1,478,045	2,905,016
Priority Queue buggy	5.4	894	894
Lin. Stack	0.2	2,690	6,896

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 the experiments. The experiments are conducted on a computer with an Intel Core-i7, 2.5 GHz processor and 8 GB RAM running Ubuntu 10.04. The first column shows the statistics of the test program, including the name and the size of benchmark. 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 transitions made. The number of states and the running time for each of the models increase with the data size.

For 3 segmented quasi queue with quasi factor 2, the verification completes in 7.2 seconds. It is much faster than the first approach presented in Section 4, where the same setting requires 130.7 seconds for the verification. Subsequently, 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 seconds and 114.5 seconds, respectively. 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. Also, it is important to note that the counterexample is produced with exploration of only part of the state space for the buggy models. The verification time is much faster for the buggy queue, which shows that our approach is effective if the quasi linearizability is not satisfied. In all test cases, our method was able to correctly verify quasi linearizability of the implementation or detect the violations.

7 Related Work

In the literature, although there exists a large body of work on formally verifying linearizability in models of data structure implementations, none of them can verify quasi linearizability. For example, Liu et al. [7, 8] use a process algebra based tool to verify that an implementation model refines a specification model – the refinement relation implies linearizability. Vechev et al. [9] use the SPIN model checker to verify linearizability in a Promela model. Cerný et al. [10] use automated abstractions together with model checking to verify linearizability properties. There also exists some work on proving linearizability by constructing mechanical proofs, often with significant manual intervention (e.g., [12, 13]).

There are also runtime verification algorithms such as Line-Up [11], which can directly check the actual source code implementation but for violations on bounded executions and deterministic linearizability. However, quasi linearizable data structures are inherently nondeterministic. For example, the `deq` operation in a quasi queue implementation may choose to return any of the first k items in a queue. To the best of our knowledge, no existing method can directly verify quasi linearizability for execution traces of unbounded length.

Besides (quasi) linearizability, there also exist many other consistency conditions for concurrent computations, including sequential consistency [20], quiescent consistency [21], and eventual consistency [22]. Some of these consistency conditions in principle may be used for checking the correctness of data structure implementations, although so far, none of them is as widely used as (quasi) linearizability. These consistency conditions do not involve quantitative aspects of the properties. We believe that it is possible to extend our refinement algorithm to verify some of these properties. However, we leave it for future work.

Outside the domain of concurrent data structures, *serializability* and *atomicity* are two popular correctness properties for concurrent programs, especially at the application level. There exists a large body of work on both static and dynamic analysis for detecting violations of such properties (e.g., [23, 24] and [25–28]). These existing methods are different from ours because they are checking different properties. Although atomicity and serializability are fairly general correctness conditions, they have been applied mostly to the correctness of shared memory accesses at the load/store instruction level. Linearizability, in contrast, defines correctness condition at the method call level. Furthermore, existing methods for checking atomicity and serializability do not deal with the quantitative aspects of the 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 manual construction of the relaxed specification model, whereas the other is fully automated, and is based on checking a relaxed version of the refinement relation between the implementation model and the specification model. We believe that the automated refinement checking algorithm can be further optimized to improve performance. For future work, we plan to incorporate advanced state space reduction techniques such as symmetry reduction and partial order reduction.

References

1. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3) (1990) 463–492
2. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2008)
3. Afek, Y., Korland, G., Yanovsky, E.: Quasi-Linearizability: Relaxed consistency for improved concurrency. In: International Conference on Principles of Distributed Systems. (2010) 395–410
4. 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. (2013)

5. Kirsch, C.M., Payer, H., Röck, H., Sokolova, A.: Performance, scalability, and semantics of concurrent fifo queues. In: International Conference on Algorithms and Architectures for Parallel Processing. (2012) 273–287
6. Payer, H., Röck, H., Kirsch, C.M., Sokolova, A.: Scalability versus semantics of concurrent fifo queues. In: ACM Symposium on Principles of Distributed Computing. (2011) 331–332
7. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. In: Proceedings of the 2nd World Congress on Formal Methods, Berlin, Heidelberg, Springer-Verlag (2009) 321–337
8. Liu, Y., Chen, W., Liu, Y., Zhang, S., Sun, J., Dong, J.S.: Verifying linearizability via optimized refinement checking. IEEE Transactions on Software Engineering (2013)
9. Vechev, M.T., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: International SPIN Workshop on Model Checking Software. (2009) 261–278
10. 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. (2010) 465–479
11. 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. (2010) 330–340
12. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: International Conference on Verification, Model Checking, and Abstract Interpretation, Berlin, Heidelberg, Springer-Verlag (2009) 335–348
13. 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, New York, NY, USA, ACM (2006) 129–136
14. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Proceedings of the 21th International Conference on Computer Aided Verification. Volume 5643 of Lecture Notes in Computer Science. (2009) 709–714
15. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Englewood Cliffs, NJ (1985)
16. Lynch, N.: Distributed Algorithms. Morgan Kaufmann (1997)
17. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics. 2nd edn. John Wiley & Sons, Inc., Publication (2004)
18. Treiber, R.K.: Systems Programming: Coping with Parallelism. Technical Report RJ 5118, IBM Almaden Research Center (1986)
19. Sun, J., Liu, Y., Dong, J.S., Chen, C.: Integrating specification and programs for system modeling and verification. In: Proceedings of the third IEEE International Symposium on Theoretical Aspects of Software Engineering. (2009) 127–135
20. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9) (1979) 690–691
21. Aspnes, J., Herlihy, M., Shavit, N.: Counting networks. J. ACM **41**(5) (1994) 1020–1048
22. Vogels, W.: Eventually consistent. Commun. ACM **52**(1) (2009) 40–44
23. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. (2003) 338–349
24. Farzan, A., Madhusudan, P.: Causal atomicity. In: International Conference on Computer Aided Verification. (2006) 315–328
25. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multithreaded programs. IEEE Trans. Software Eng. **32**(2) (2006) 93–110
26. Farzan, A., Madhusudan, P.: Monitoring atomicity in concurrent programs. In: International Conference on Computer Aided Verification. (2008) 52–65
27. Sadowski, C., Freund, S.N., Flanagan, C.: Singletrack: A dynamic determinism checker for multithreaded programs. In: European Symposium on Programming. (2009) 394–409
28. 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. (2010) 328–342