CIT. Journal of Computing and Information Technology, Vol. 33, No. 4, December 2025, 219–229
doi: 10.20532/cit.2025.1006091

Specifying Relaxed Concurrent Data Structures Using NADTs

Jie Peng and Tangliu Wen

School of Information Engineering, Gannan University of Science and Technology, Ganzhou, China

To improve performance scalability of concurrent data structures, one solution is to relax their sequential semantics. While a variety of specification approaches focus on characterizing the relaxed semantics, client-side reasoning using the current methodologies is difficult. We employ nondeterministic abstract data types (NADTs) for the first time to specify the relaxed concurrent data structures, and as instantiations of our specification approach, we propose new correctness criteria of out-of-order queues and stacks. We further prove the relaxation equivalence of the out-of-order dequeue and enqueue operations. Our specification approach is intuitive and generic, and can provide clients with explicit interfaces. As a demonstration of our approach, we specify and verify the k-segment queue.

ACM CCS (2012) Classification: Theory of computation → Design and analysis of algorithms → Concurrent algorithms

Keywords: relaxed concurrent data structures, specification, nondeterministic abstract data type

1. Introduction

With the continuous development of multicore processors and systems, it has become increasingly important to design and implement high-concurrency data structures in order to efficiently utilize these resources [1, 2]. Correctness conditions for concurrent data structures usually equire that each execution of a concurrent data structure is equivalent to a sequential execution of its specification model (as its sequential semantics). This equivalent relationship is formally captured by consistency conditions, such as linearizability [3].

The sequential semantics of concurrent data structures inevitably lead to memory contention in parallel environments, thus limiting scalability [4–6]. One solution is to relax the sequential semantics of concurrent data structures [7, 8]. For example, a relaxed semantics of a standard "FIFO" queue allows each dequeue method to remove any of the k elements nearest to the head instead of just the head, and allows each enqueue method to insert an element into the position which is at most k positions away from the tail. In sequential environments, this relaxed semantics offers no performance improvement for the implementations of queues. However, in parallel environments, such relaxed semantics can reduce data contention. For example, multiple concurrent dequeue (enqueue) operations, which originally contend for access to a single head element (the tail position), now compete for the first k elements (the last k positions). Thus, relaxing the sequential semantics facilitates the design and implementation of higher-performance concurrent data structures [9–13]. Recently, numerous implementations of these relaxed data structures (also called relaxed concurrent data structures) have been proposed.

The existing work in [7–8] provides quantitative approaches to formally describe such relaxed semantics. Relaxing the data structure corresponds to defining a bounded distance away from the standard sequential specification. For example, Henzinger *et al.* formalized and generalized the notion of relaxations, and characterized two generic instances: out-of-order and stuttering relaxations. However, some relaxation mechanisms of concurrent data structures, such as local-thread relaxation, are difficult to quantify. Furthermore, these speci-

fication methods do not provide users with explicit specification interfaces, preventing users from directly leveraging these specifications to reason about client programs.

Nondeterministic abstract data types (NADTs) allow operations to have multiple possible outcomes for the same input parameters. In this paper, we extend existing methodologies for ADT implementation in sequential environments, and use NADTs to depict random behavior of relaxed concurrent data structures. We employe NADTs for the first time to characterize the relaxed semantics of concurrent data structures, such as out-of-order and local-thread relaxations. We further prove that the out-of-order relaxations of dequeue and enqueue operations have the same impact on causing "error bound". The out-of-order stacks do not possess analogous properties. Based on the above discovery, we propose new correctness criterions of out-of-order queues and stacks. As a demonstration of our approach, we specify and verify the k-segment queue. Our specification approach is intuitive and generic, and can provide clients with explicit interfaces. Clients do not need to know the implementation details of concurrent data structures and can use the NADTs interfaces to reason about their programs.

The main contributions of this paper are:

1. present the specification framework based on NADTs for relaxed concurrent data structures
2. prove the relaxation equivalence of the out-of-order dequeue and enqueue operations
3. specify and verify the k-segment queue.

The structure of this paper is as follows. In Section 2, we recall the definition of linearizability. In Section 3, we formalize nondeterministic abstract data types, and illustrate the specification framework on two generic instances, out-of-order and local-thread relaxations. In Section 4, we prove that the relaxation equivalence of the out-of-order dequeue and enqueue operations and show that the out-of-order pop and push operations do not hold the analogous property. In Section 5, we present new correctness criteria for out-of-order queues and stacks. In Section 6, we specify and verify the k-segment queue. Finally, we discuss related work and conclude in Section 7 and 8.

2. Linearizability

In this section, we introduce basic notations and review the definition of linearizability [3]. We refer to an execution of a method as an *operation*. We denote an execution as a finite sequence of totally ordered atomic events. We represent the calling of a method by an invocation event, and the return of a method by a response event. An execution of a method starts with the invocation event, executes its internal atomic events until the final response event.

A history of a concurrent data structure is a sequence of its invocation and response events generated in an execution. An invocation event matches a response event if they belong to the same operation. A history is sequential if every invocation event, except possibly the last, is immediately followed by its matching response event. A sequential history of a concurrent data structure is legal if its corresponding sequential execution satisfies the sequential specification of the concurrent data structure.

A history is complete if every invocation event has a matching response event. An invocation event is pending in a history if there is no matching response event to it. For an incomplete history H, a completion of H is a complete history gained by adding some matching response events to the end of H and removing some pending invocation events within H. Let $Compl(H)$ be the set of all completions of the history H. For any two operations op_1 and op_2, we say that op_1 precedes op_2 in a history, if the response event of op_1 precedes the invocation event of op_2.

A history H of a concurrent data structure is linearizable with respect to its sequential specification if there exists a complete history $C \in Compl(H)$ and a legal sequential history S such that (1) S is a permutation of C; (2) for any two operations op_1, op_2, if in C, op_1 precedes op_2, then in S, op_1 also precedes op_2. S is called a *linearization* of H. A concurrent data structure is linearizable with respect to its sequential specification if every history of the concurrent data structure is linearizable with respect to the sequential specification.

3. Characterizing Relaxations Using NADTs

In this section, we formalize nondeterministic abstract data types (NADTs). We use NADTs as the specification models of relaxed concurrent data structures and illustrate the specification framework on two generic instances, out-of-order and local-thread relaxations.

3.1. Nondeterministic Abstract Data Types

We use the model-based way to define nondeterministic abstract data types (NADTs), where a NADT is considered as a set of abstract values together with a set of atomic methods; the methods are specified by defining how they affect the abstract values. Behaviors of the methods in the NADTs are nondeterministic, *i.e.*, A method may return different results each time it is called under a specific pair of state and input.

Definition 1. A nondeterministic abstract data type is a tuple,(*AState*, σ, *Aop*, *Input*, *Output*), where *AState* is a set of states; $\sigma \in$ *AState* is the initial state; *Aop* is a set of methods; *Input* is a set of input values; *Output* is a set of output values; each method op $\in$ *Aop* is a mapping op: *AState* $\times$ *Input* $\nrightarrow$ *P*(*AState* $\times$ *Output*).

For example, $op(\sigma_0, in) = \{(\sigma_1, ret_1), ..., (\sigma_n, ret_n)\}$ denotes that the result of applying the operation (or function) *op* to an input *in* and a state σ_0 is the set $\{(\sigma_1, ret_1), ..., (\sigma_n, ret_n)\}$.

Let $(\sigma, in)op(\sigma', ret)$ denote that the sequential execution of *op* started in a state σ with an input *in* terminates in a state σ' with an output *ret*. Let *dom*(*op*) denote the domain of the method *op*. $(\sigma, in)op(\sigma', ret)$ is a legal execution if $(\sigma, in) \in dom(op)$ and $(\sigma', ret) \in op(\sigma, in)$. A sequential execution $(\sigma_0, i_1)op_1(\sigma_1, o_1)... (\sigma_{n-1}, i_n)op_n(\sigma_n, o_n)$ is legal if $(\sigma_{i-1}, i_i) \in dom(op_i)$ and $(\sigma_i, o_i) \in op_i(\sigma_{i-1}, i_i)$, for each $1 \le i \le n$.

3.2. Out-of-order Relaxation

The out-of-order relaxation allows relaxed operations to deviate from the sequential order. For example, an out-of-order queue could allow each enqueue operation to insert an element into any of the positions which are at most k_1 away from the tail, and each dequeue operation to remove any of the first k_2 elements instead of just the head. The nondeterministic abstract queue, denoted (k_1, k_2)-queue, is used to characterize the relaxed semantics and is defined formally as follows.

$$enqueue(s, e) = \left\{ \left(s_1 \frown e \frown s_2, null \right) \mid \exists s_1, s_2 .\ s_1 \frown s_2 = s \wedge |s_1| \le k_1 \right\} \quad (1)$$

$$dequeue(s, \varepsilon) = \begin{cases} (s, empty), |s| = 0 \\ \left\{ (s', e) \mid \exists s_1, s_2 .\ s_1 \frown s_2 = s' \wedge s_1 \frown e \frown s_2 = s \wedge |s_2| \le k_2 \right\}, |s| > 0 \end{cases} \quad (2)$$

Null and ε denote that the enqueue and dequeue methods have no return and input values, respectively. When $k_1 = 0$ and $k_2 = 0$, both S_1 and S_2 are empty sequences. Thus (0, 0)-queue is a standard "FIFO" queue.

We further relax the semantics of dequeue operations and allow a dequeue operation to return *empty* even when the queue is not empty. Formally, the relaxed dequeue method is specified as follows. When the current size k_3 of the queue is very small, the clients may accept the value *empty*.

$$dequeue(s, \varepsilon) = \begin{cases} \left\{ (s', e) \mid \exists s_1, s_2 .\ s_1 \frown s_2 = s' \wedge s_1 \frown e \frown s_2 = s \wedge |s_2| \le k_2 \right\} \\ \cup \left. (s, empty) \right| |s| \le k_3 \end{cases} \quad (3)$$

In a similar way as for queue, we can define an out-of-order stack. The out-of-order relaxed stack allows each push method to insert an element at most k1 away from the top, and a pop method to remove an element at most k_2 away from the top. The nondeterministic abstract stack, denoted (k_1, k_2)-stack, is used to characterize the relaxed semantics and is defined formally as follows.

$$push(s, e) = \left\{ (s_1 \frown e \frown s_2, null) \mid \exists s_1, s_2 .\ s_1 \frown s_2 = s \wedge |s_2| \le k_1 \right\} \quad (4)$$

$$pop(s, \varepsilon) = o \quad (5)$$

$$\begin{cases} (s, empty), |s| = 0 \\ \left\{ (s', e) \mid \exists s_1, s_2 .\ s_1 \frown s_2 = s' \wedge s_1 \frown e \frown s_2 = s \wedge |s_2| \le k_2 \right\}, |s| > 0 \end{cases}$$

3.3. Local-thread Relaxation

Local-linearizability (called local-thread relaxation) is a relaxation mechanism that is applicable to container-type concurrent data structures like pools, queues, and stacks. For a local-linearizable queue, the induced history of a thread T is a projection of a history to the enqueue-operations of T combined with all dequeue-operations that dequeue elements enqueued by T. The local-linearizability only requires that each thread-induced history is linearizable. For example, a local-linearizable concurrent queue only requires that elements inserted by a single thread satisfy the FIFO semantics, rather than enforcing this ordering for all elements in the entire queue.

We assume that the elements enqueued onto the queue are unique. Let $s \lceil t$ denote the maximal subsequence of s consisting of the elements enqueued by the thread t. let $s-e$ denote the subsequence of s by deleting the element e. $First(s)$ denote the first element of the sequence s. The sequential semantics of a local-linearizable queue is specified by the following the nondeterministic abstract queue.

$$enqueue(s,e)_t = \left\{(s_1 \frown e \frown s_2,\ null) | \exists s_1, s_2.\ s_1 \frown s_2 = s \wedge s_2 \lceil t = null\right\} \quad (6)$$

$$dequeue(s,\varepsilon) = \left\{(s-e,\ e) | \exists thread\ t.\ e = First(s \lceil t)\right\} \quad (7)$$

Each enqueue operation $enqueue(s,\ e)_t$ of a thread t inserts an element into the tail of the sequence which consists of the elements enqueued by t. Each dequeue operation randomly selects a thread and removes the head element of the thread's sequence (*i.e.*, $s \lceil t$). Thus, the nondeterministic abstract queue guarantees that the elements enqueued by a single thread are dequeued in FIFO order.

4. Equivalence of Operations

In this section, we will show that the relaxations of enqueue and dequeue operations have an equivalent impact on causing "error bound", and push and pop operations do not hold the analogous property.

4.1. Relaxation Equivalence of Out-of-order Enqueue and Dequeue Operations

The following lemma shows that each dequeue operation of the (k_1, k_2)-queue removes at least the $(k_1 + k_2 + 1)$-th earliest-inserted element.

Lemma 1. For a legal sequential execution of a (k_1, k_2)-queue, each dequeue operation can remove anyone of the $k_1 + k_2 + 1$ oldest elements in the current queue and cannot any element which is enqueued later than the $(k_1 + k_2 + 1)$-th earliest-inserted element in the current queue.

The proof is written in a hierarchically structured style.

1. The first dequeue operation of the execution can remove anyone of the $k_1 + k_2 + 1$ oldest elements in the current queue.

Proof. Assume an element e which is inserted by the x-th enqueue operation before the first dequeue operation. If $x \leq k_1$, the number of elements in the queue is less than k_1 when e is inserted into the queue. By the relaxation factor k_1 of enqueue operations, e can be inserted into the head of the queue. Thus, the first dequeue operation can remove e. If $k_2 < x \leq k_1 + k_2 + 1$, then the number of elements in the queue is less than or equals to $k_1 + k_2$ when e is inserted into the queue. By the relaxation factor k_1 of enqueue operations, e can be inserted into the position which is k_1 away from the tail of the queue. Because the size of the queue is less than or equals to $k_1 + k_2 + 1$, the element e's distance from the head will not exceed k_2. By the relaxation factor k_2 of dequeue operations, the first dequeue operation can also remove e. If $k_1 + k_2 + 1 < x$, the element e's distance from the head must exceed k_2. In this case, the first dequeue operation cannot remove the element.

2. All possible states of the queue after the first dequeue operation are the states after the execution of the rest of enqueue operations in the original order.

Proof. Assume the element removed by the first dequeue operation is e_1 and the state before the enqueue operation (which inserted the element e_1) is σ. Assume the element e_2 is inserted first after the element e_1 is inserted. The element e_1 can insert the position which is k_1 away from

the tail or the head of the queue (if the length of the queue is smaller than k_1).

The element e_2 can be inserted into any position behind e_1. Thus, all possible states after e_1 are removed equals the possible states after e_2 is inserted from the state σ.

3. Q.E.D.

Proof. By 1 and 2, we can establish the theorem.

Remark. For the (k_1, k_2)-queue and the (k_3, k_4)-queue, if $k_1 + k_2 = k_3 + k_4$, then the most recent element which their dequeue operations can remove is the $(k_1 + k_2 + 1)$-th (or $(k_3 + k_4 + 1)$-th) earliest-inserted element. For example, the dequeue operations in both $(k, 0)$-queue and $(0, k)$-queue remove at least the $(k + 1)$-th earliest-inserted element. Thus, the relaxations of dequeue and enqueue operations have an equivalent impact on causing "error bound" (a bound on the ranking of the dequeued elements based on the enqueue order, not the bound on the distance of a dequeued element from the head).

The following theorem further shows that if $k_1 + k_2 = k_3 + k_4$, then the (k_1, k_2)-queue and the (k_3, k_4)-queue are two equivalent specification models w.r.t. linearizability.

Theorem 1. For any two abstract nondeterministic queues (k_1, k_2)-queue and (k_3, k_4)-queue with $k_1 + k_2 = k_3 + k_4$, if a relaxed concurrent queue is linearizable w.r.t. the (k_1, k_2)-queue, then it is also linearizable w.r.t. the (k_3, k_4)-queue, and vice versa.

Proof. By the Lemma 1, if a sequential history is legal w.r.t the (k_1, k_2)-queue, it is also legal w.r.t the (k_3, k_4)-queue, and vice versa. Because the concurrent queue is linearizable w.r.t the (k_1, k_2)-queue, for any history H of the concurrent queue, there exists a legal sequential history S of the (k_1, k_2)-queue such that S is a linearization of H. S is also legal w.r.t. the (k_3, k_4)-queue. Thus, the history H is also linearizable w.r.t. the (k_3, k_4)-queue.

For example, consider the following sequential history, where we directly use the operation to replace its invocation and response events; *enqueue*(x) denotes an enqueue operation with an input parameter x; *dequeue*(y) denotes a dequeue operation with a return value y.

enqueue(a); *enqueue*(b);
enqueue(c); *dequeue*(c)

The sequential history is legal w.r.t. the (2, 0)-queue. Because the relaxation factor of the enqueue method is 2, c can be inserted into the head of the queue. Thus, a possible state is *bac* before the dequeue operation begins to execute. Obviously, the dequeue operation can remove the head element c. The sequential history is also legal w.r.t. (0, 2)-queue. Because the relaxation factor of the enqueue operation is 0, the state is *cba* (c is the tail element) after c is inserted. Because the relaxation factor of the dequeue operation is, the enqueue operation can remove the tail element c. y similar analysis, we can show the sequential history is also legal w.r.t. the (1, 1)-queue.

4.2. Non-equivalence of Out-of-order Push and Pop Operations

In this subsection, we show that the analogues of Lemma1 and Theorem 1 do not hold for out-of-order stacks. The oldest element which a pop operation of the (k_1, k_2)-stack can pop is not the $(k_1 + k_2 + 1)$-th latest element. In fact, if the relaxation factor of push operations $k_1 > 0$, a pop operation of the (k_1, k_2)-stack can pop the oldest element in the stack. For example, consider the following sequential execution:

push(a); *push*(b); *push*(c); *push*(d); *pop*( )

For the (1, 1)-stack, the pop operation can pop anyone of the elements a, b, c, d. For example, if the elements b, c, d are not inserted at the top of the stack (by the relaxation factor 1 of push operations), the element a can remain on top of the stack, and the state of the stack after the *push*(d) operation is *bcda* (a is on top). Obviously, the pop operation can pop element a.

(k_1, k_2)-stack and (k_3, k_4)-stack are not two equivalent specification models under the linearizability correctness criterion, even if $k_1 + k_2 = k_3 + k_4$. Consider the above execution. For the (0, 2)-stack, the state after the *push*(d) operation is *abcd* (d is on top, by the relaxation factor 0 of push operations). The pop operation can pop one of the elements b, c, d, but cannot

pop the element *a* (by the relaxation factor 2 of pop operations). Thus, the (1, 1)-stack is not equivalent to (0, 2)-stack under linearizability correctness criterion.

5. Correctness

We adopt nondeterministic abstract data types as the specification models for relaxed concurrent data structures and employ linearizability as the correctness criterion for these data structures. If a sequential history is legal w.r.t. a nondeterministic abstract type, then it is also legal w.r.t. a more nondeterministic version of the abstract type. Thus, if a concurrent implementation is linearizable w.r.t. a nondeterministic abstract type then it is also linearizable w.r.t. the more nondeterministic abstract type. For example, if a relaxed concurrent stack is linearizable w.r.t. the (1, 1)-stack, then it is also linearizable w.r.t. the (2, 3)-stack. Obviously, the former is a more precise specification for the relaxed concurrent stack. To obtain a precise specification based on NADTs, an intuitive method is to specify the relaxation bound of the relaxed operations. The following definition requires that for a relaxed concurrent implementation of the (k_1, k_2)-stack, k_1 and k_2 are the maximal relaxation factors of push and pop methods of the implementation, respectively.

Definition 2. A relaxed concurrent stack is an implementation of the ((k_1, k_2)-stack if it is linearizable w.r.t. the (k_1, k_2)-stack and for any $x < k_1$ or $y < k_2$, it is not linearizable w.r.t. the (x, y)-stack.

By Theorem 1, if a relaxed concurrent queue is linearizable w.r.t (k_1, k_2)-queue and $k_1 + k_2 = k_3 + k_4$ (for two positive integers k_3, k_4), then it is also linearizable w.r.t (k_3, k_4)-queue. Thus, unlike relaxed concurrent stacks, there do not exist the maximal relaxation factors of enqueue and dequeue operations to specify relaxed concurrent queues.

For a standard queue, an enqueue operation inserts elements at the tail of the queue, while a dequeue operation removes the elements at the head. Under FIFO semantics, the dequeue operation always deletes the oldest element in the queue. For the relaxed concurrent queues, clients probably do not care where the elements are inserted, but the precision of the dequeue operations-how old elements they can remove at least. The following definition requires that for an implementation of the (k_1, k_2)-queue, each dequeue operation removes at least the $(k_1 + k_2 + 1)$-th earliest-inserted element.

Definition 3. A relaxed concurrent queue is an implementation of the (k_1, k_2)-queue if it is linearizable w.r.t. the (k_1, k_2)-queue and for any two positive integers x, y. if $x + y < k_1 + k_2$, then it is not linearizable w.r.t. the (x, y)-queue.

Similarly, for the relaxed concurrent stacks, clients are likely most concerned about how recent elements the pop operations can remove at least. Unlike the relaxed concurrent queues, if the relaxation factor of the push method is more than 1, the pop operation can pop the oldest element in the stack (as mentioned in Section 4). Thus, if clients require that each pop operation removes at least the k-th most recent element (some positive integer k), then relaxation factors of push methods should be 0, *i.e.*, the relaxed concurrent stacks should be an implementation of the $(0, k-1)$-stack.

6. Specifying and Verifying the K-segment Relaxed Queue

The pseudo code of the k-segment relaxed queue [8] is depicted in Figure 1. The queue maintains a linked list. Each node (also called segment) has a value field *array*, holding an array of length k, and a next field *next*, linking nodes in the list. Using a dummy node ensures that Head and Tail are always non-null. Head always points to the dummy node. Every slot of the array is initialized with a value *null*, signaling no element present.

An enqueue operation tries to insert an element in the array of the last segment. Each element has two fields, the actual value *val* and a Boolean marker *del*, which indicates if it has been dequeued.

The methods *init*, *try_add_LastSegment* and *try_remove_first_segment* implement the operations of segments atomically. *try_add_LastSegment* (lastSegment) creates and adds a new last segment only if the current last segment is the argument *lastSegment*. The method *try_remove_first_segment* (firstSegment) removes

the first segment only if the current first segment is the argument *firstSegment*.

If the queue is not empty, the enqueue method iterates over the tail segment in random order (line 26) to search for an empty slot; When it finds an empty slot it performs a CAS operation attempting to insert the new element into the empty slot (line 31). If no empty slot is found in the current tail segment (implying that the segment is full), then it tries to add a new tail segment to the list (line 33) and then retries. If the queue is empty, it also attempts to add a new tail segment to the list (line 35) and then retries.

If the queue is empty (line 39), the dequeue method returns *empty* (line 40). Otherwise, the dequeue method iterates over the array of the first segment in random order (line 41). If the item is null in the current slot, the flag *hadNullValue* is set to *true*, and then starts the next iteration (line 45). Otherwise, the method performs a CAS operation on its marker field in order to delete it (line 46), and it returns the value of the item if the deletion is successful (line 47). If no item is deleted after completing the iteration, and there was an empty cell in the current segment (line 48), the dequeue method returns *empty* (line 49). If there is not any empty slot in the current segment (implying that all items of the current segment have already been deleted), then it attempts to remove this segment from the linked list (line 50) and retries.

The specification model of the relaxed queue is defined as follows.

$$enqueue(s, e) = (s \frown e,\ null) \tag{8}$$

$$dequeue(s, \varepsilon) = \tag{9}$$
$$\begin{cases} \left\{ (s',\ e) \mid \exists s_1,\ s_2.\ s_1 \frown s_2 = s' \wedge s_1 \frown e \frown s_2 = s \wedge |s_2| \le k-1 \right\} \\ \cup\ (s,\ empty) \mid |s| \le k-1 \end{cases}$$

We now prove that the k-segment queue is correct w.r.t. the above specification model. In the specification model, the relaxed factors of the enqueue and queue methods are 0 and k–1, respectively. Thus, we will prove that each dequeue operation of the k-segment queue at least removes the kth oldest element in the current queue, and if a dequeue operation returns empty, then the number of elements in the queue is at most k–1.

Proposition 1. The k-segment relaxed queue is a correct implementation of the above specification model (0, k–1)-queue.

Proof. Without loss of generality, we assume that the values enqueued onto the queue are unique. The linearization point of an enqueue operation is the successful CAS action of line 29 (successful insertion). The linearization point of a non-empty dequeue operation is the successful CAS action of line 46 (successful removal). The linearization point of the dequeue operation which returns *empty* in line 40 is the reading action of line 37 in the lastiteration. At this point, the queue is empty. The dequeue operation which returns *empty* in line 49 is relaxed (*i.e.*, it may return empty in the not-empty state). The linearization point of the dequeue operation is the action of line 41 in the last iteration.

We will prove that at this point, the number of elements in the queue is at most k–1. According to the linearization points, an element x is enqueued earlier than an element y (*i.e.*, x is older than y) if the linearization point of x's corresponding enqueue operation is earlier than the one of y's corresponding enqueue operation. The correctness argument is based on the following facts.

1. *Each enqueue operation only enqueues an item into the queue; An item is enqueued exactly once.*

This is because each enqueue operation has a unique action (at line 29) which modifies an empty slot to contain an item, and every item is unique by our assumption.

2. *An item is logically deleted at most once. Each dequeue operation only logically deletes an item.*

This is because each dequeue operation has a unique action (at line 29) which modifies the field del of an item to true. Once the field of an item is modified, no other actions will modify it again.

3. *An item inserted by an enqueue operation is in a reachable segment from Head if it is not deleted by a dequeue operation.*

Tail always is reachable from Head. There is no cycle in this linked list. Every element is inserted into the tail segment at its linearization point. A segment is removed from the list only if every element of the segment has been removed.

```
1  global  Head, Tail;
2  void init()
3      Array array = new Array(k);
4      array.fill(⊤);
5      Node dummySegment = new Node(array);
6      Head = Tail = dummySegment;
7
8  try_add_LastSegment(lastSegment)
9     Array newArray = new Array(k);
10    newArray.fill(null);
11    Node newSegment = new Node(newArray);
12    if(lastSegment.next == null)
13        if(CAS(lastSegment.next, null, newSegment)
14            CAS(Tail, lastSegment, newSegment);
15
16 try_remove_first_segment(firstSegment)
17    head_old = Head;
18    if(head_old.next == firstSegment)
19        CAS(Head, head_old, firstSegment)
20
21 void enq(value)
22    Item newItem = new Item(value, false);
23    while true:
24        Node lastSegment = Tail;
25        if lastSegment != Head:
26            for i in shuffle([1,...,k]):
27                if(lastSegment.array[i] != null)
28                    continue;
29                if(CAS(lastSegment.array[index], null, newItem))
30                    return;
31            try_add_LastSegment(lastSegment);
32       else
33            try_add_LastSegment(lastSegment);
34
35 value deq( )
36    while true:
37       Node firstSegment = Head.next;
38       boolean hadNullValue = false;
39       if firstSegment == null:
40           return empty;
41       for i in shuffle([1,...,k]):
42           item = firstSegment.array[i];
43           if item == null:
44               hadNullValue = true;
45               continue;
46           if CAS(item.del, false, true):
47                return item.val;
48       if hadNullValue:
49           return empty;
50       try_remove_first_segment(firstSegment);
```

Figure 1. The k-segment relaxed queue.

4. *The dequeue operation always deletes at least the kth oldest element of the linked list at its linearization point.*

Every element is enqueued into the tail segment at its corresponding linearization point, and a new tail segment is added only if every slot of the old tail segment has been inserted an element. Thus, the elements of preceding segments are enqueued earlier than the ones of subsequent segments in the list. The dequeue operation always deletes an element of the second segment randomly (the first segment is a dummy segment). Thus, the dequeue operation always deletes at least the kth oldest element of the queue at its linearization point.

5. *The dequeue method which returns empty in line 49 is relaxed. The number of elements in the queue is at most k−1 before its last iteration. During and after the last iteration, the number of the elements may be more than k.*

Because *hadNullvalue* is true after the last iteration of the empty dequeue operation, the second segment has empty slots before its last iteration. Thus, the second segment is the only segment except the dummy segment and the queue has at most $k-1$ elements before the dequeue operation executes. Empty slots may have been inserted into items and a new segment may have been added after empty slots had been scanned by the dequeue operation. Thus, during and after the last iteration, the number of elements in the queue may be more than k.

7. Related Work

The exploration of relaxed concurrent data structures has garnered significant attention due to their potential for enhanced scalability and performance in multi-threaded environments. A substantial number of instances of relaxed concurrent data structures have been designed and implemented [15–18]. There are two main ways to specify relaxed concurrent data structures formally: relaxing linearizability and relaxing their sequential specification. Talmage *et al.* [19] explored the relationship between them, showed that in many cases they can yield equivalent sets of allowed concurrent behaviors.

The works on relaxing linearizability [14, 20–22] proposed weaker consistency conditions than linearizability, such as local-linearizability, interval-linearizability, intermediate value linearizability, visibility relaxation. For example, Rinberg *et al.* [20] proposed intermediate value linearizability, that relaxes linearizability to allow read operations to return any value that is bounded between two return values that are legal under linearizability. Castañeda *et al.* [21] proposed interval-linearizability that relaxes linearizability to allow every operation takes effect "continuously" in an interval of time, instead of instantaneously at a single point. These weaker consistency conditions are able to precisely specify the relaxed behavior of concurrent data structures. However, they do not provide users with explicit specification interfaces, and it is unclear how to use them to reason about client-side programs.

Our work is closely related to works on relaxing the sequential semantics of concurrent data structures, such as the quantitative framework for relaxation in [7], and quasi-linearizability in [8]. Henzinger *et al.* proposed a quantitative framework to formally specify relaxed sequential specification. The relaxed semantics is introduced by defining semantical distances from the deterministic sequential specification. For instance, the k-relaxed sequential specification contains all "wrong" executions within distance k from the original specification. Quasi-linearizability also defines quantitative relaxations through a distance function from valid sequential executions. The quantitative relaxation of quasi-linearizability is syntactic and less expressive than the quantitative framework of [7]. Henzinger *et al.* apply the quantitative framework on two simple yet generic relaxation mechanisms: out-of-order and stuttering relaxation. However, with regard to the quantitative frameworks, some relaxation mechanisms are difficult to quantify. For a relaxed concurrent queue which employs thread-local relaxation, only the elements which are enqueued by the same thread satisfies the "FIFO" semantics (as mentioned in section 3). It is impossible to assign a meaningful quantitative value (the maximal distance from the head of a dequeued element) for this type of queue. Unlike the quantitative framework, our method uses NADTs to specify the relaxed semantics directly.

8. Conclusion

In this work, we present the specification framework based on NADTs for relaxed semantics of concurrent data structures. Our specification methodology enables clients to use NADTs interfaces to reason about their programs, not to need to know the implementation details of relaxed concurrent data structures. To the best of our knowledge, we are the first to use NADTs for characterizing relaxed semantics of relaxed concurrent data structures.

Our correctness criterion based on linearizability can ensure observational refinement [23] and the relaxation bound. A relaxed concurrent data structure can be viewed as an implementation of a NADT. Like in the sequential environments, data abstraction in the concurrent environments should also ensure observational equivalence. In the future, we will present a new correctness criterion for relating a NADT and its concurrent implementation, which can ensure observational equivalence.

Conflict of Interest

The authors declare no conflict of interest.

Data Availability

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

Funding

This research was funded by Science and Technology Research Project of Jiangxi Province Educational Department [No.GJJ2203609].

References

[1] N. Shavit, "Data structures in the multicore age", *Communications of the ACM*, vol. 54, no. 3, pp. 76–84, 2011. https://doi.org/10.1145/1897852.1897873

[2] A. Castañeda and S. Rajsbaum, "Recent Advances on Principles of Concurrent Data Structures", *Communications of the ACM*, vol. 67, no. 8, pp. 45–46, 2024. https://doi.org/10.1145/365329

[3] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990. https://doi.org/10.1145/78969.78972

[4] H. Attiya *et al.*, "Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot be Eliminated", *ACM SIGPLAN Notices*, vol. 46, no. 1, pp. 487–498, 2011. https://doi.org/10.1145/1925844.1926442

[5] J. Wang *et al.*, "Improved Time Bounds for Linearizable Implementations of Abstract Data Types", in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IEEE Computer Society*, 2014, pp. 691–701. https://doi.org/10.1109/IPDPS.2014.77

[6] E. Talmage and J. L. Welch, "Improving Average Performance by Relaxing Distributed Data Structures", in *Proceedings of the 28th International Symposium on Distributed Computing*, 2014, pp. 691–701. https://doi.org/10.1109/IPDPS.2014.77

[7] T. A. Henzinger *et al.*, "Quantitative Relaxation of Concurrent Data Structures", in *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2013 pp. 317–328. https://doi.org/10.1145/2429069.2429109

[8] Y. Afek *et al.*, "Quasi-linearizability: Relaxed Consistency for Improved Concurrency", in *Proc. of the Conference on Principles of Distributed Systems (OPODIS)*, 2010, pp. 395–410. https://doi.org/10.1007/978-3-642-17653-1_29

[9] K. von Geijer *et al.*, "Balanced Allocations over Efficient Queues: A Fast Relaxed FIFO Queue", in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2025, pp. 382–395. https: // doi.org /10.1145/ 3710848.3710892

[10] A. Castañeda and M. Piña, "Read/write Fence-free Work-stealing with Multiplicity", *Journal of Parallel and Distributed Computing*, vol. 186, no. 104816, 2024. https://doi.org/10.1016/j.jpdc.2023.104816

[11] J. Ko, "A Lock-free Binary Trie", *IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*, pp. 163–174, 2024. https://doi.org/10.1109/ICDCS 60910.2024.00024

[12] A. Rukundo and A. Atalar, "Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework", 33rd International Symposium on Distributed Computing, pp. 31:1–31:15, 2019. https://doi.org/10.4230/LIPIcs.DISC.2019.31

[13] A. A. Paznikov and A. D. Anenkov, "Implementation and Analysis of Distributed Relaxed Concurrent Queues in Remote Memory Access Model", *Procedia Computer Science*, vol. 150, pp. 654–662, 2019. https://doi.org/10.1016/j.procs.2019.02.101

[14] A. Haas *et al.*, "Local linearizability for Concurrent Container-type Data Structures", *Leibniz International Proceedings in Informatics*, 2016. https://doi.org/10.4230/LIPIcs.CONCUR.2016.6

[15] S. Bouhenni *et al.*, "Efficient Parallel Edge-centric Approach for Relaxed Graph Pattern Matching", *The Journal of Supercomputing*, vol. 78, no. 2, pp. 1642–1671, 2022. https://doi.org/10.1007/s11227-021-03938-7

[16] A. V. Tabakov and A. A. Paznikov "Using Relaxed Concurrent Data Structures for Contention Minimization in Multithreaded MPI Programs", *Journal of Physics: Conference Series*, vol. 1399, no. 3, 2019. https://iopscience.iop.org/article/10.1088/1742-6596/1399/3/033037/meta

[17] K. von Geijer and P. Tsigas, "How to Relax Instantly: Elastic Relaxation of Concurrent Data Structures", in European Conference on Parallel Processing, pp. 119–133, 2024. https:// doi.org/10.1007/978-3-031-69583-4_9

[18] A. Singer *et al.*, "MultiQueue-Based FPGA Routing: Relaxed A* Priority Ordering for Improved Parallelism", in *Proc. of the 23rd International Conference on Field-Programmable Technology*, 2024. https://infoscience.epfl.ch/handle/20.500.14299/242260

[19] E. Talmage and J. L. Welch, "Relaxed Data Types as Consistency Conditions", *Algorithms*, vol. 11, no. 5(61), 2018. https://doi.org/10.3390/a11050061

[20] A. Rinberg and I. Keidar, "Intermediate Value Linearizability: A Quantitative Correctness Criterion", *Journal of the ACM*, vol. 70, no. 2(17), pp. 1–21, 2023. https://doi.org /10.1145/3584699

[21] A. Castañeda *et al.*, "Unifying Concurrent Objects and Distributed Tasks: Interval-linearizability", *Journal of the ACM*, vol. 65, no. 6(45), pp. 1–42, 2018. https:// doi.org/10.1145/3266457

[22] J. Öhman and A. Nanevski, "Visibility Reasoning for Concurrent Snapshot Algorithms", in *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL(33), pp. 1–30, 2022. https://doi.org/10.1145/3498694

[23] I. Filipović *et al.*, "Abstraction for Concurrent Objects", *Theoretical Computer Science*, vol. 411, no. 51–52, pp. 4379–4398, 2010.

Received: August 2025
Revised: September 2025
Accepted: September 2025

Contact addresses:
Jie Peng
School of Information Engineering
Gannan University of Science and Technology
Ganzhou
China
e-mail: pengjie@gnust.edu.cn

Tangliu Wen
School of Information Engineering
Gannan University of Science and Technology
Ganzhou
China
e-mail: wqtlglk@163.com

Jie Peng received a master's degree in Computer Application Technology from Jiangxi University of Science and Technology in 2019. Currently, she is working as an associate professor and Director of the Computer Teaching and Research Office at Gannan University of Science and Technology. Her main research directions include software engineering, AI and big data.

Tangliu Wen obtained bachelor's degree in information technology from Nanchang University, his master's degree in Computer Application Technology from Jiangxi University of Science and Technology, and his PhD in Computer Software and Theory from Wuhan University in 2019. Presently, he is working as an associate professor and Director of the E-Commerce Teaching and Research Office at Gannan University of Science and Technology. His key areas of research include formal software analysis and verification, AI and big data, and intelligent information systems.