Quiescent Consistency: Defining and Verifying Relaxed Linearizability

John Derrick[1], Brijesh Dongol[1], Gerhard Schellhorn[2], Bogdan Tofan[2], Oleg Travkin[3], and Heike Wehrheim[3]

[1]Department of Computing, University of Sheffield, Sheffield, UK
[2]Universität Augsburg, Institut für Informatik, 86135 Augsburg, Germany
[3]Universität Paderborn, Institut für Informatik, 33098 Paderborn, Germany

Abstract. Concurrent data structures like stacks, sets or queues need to be highly optimized to provide large degrees of parallelism with reduced contention. Linearizability, a key consistency condition for concurrent objects, sometimes limits the potential for optimization. Hence algorithm designers have started to build concurrent data structures that are not linearizable but only satisfy relaxed consistency requirements.
In this paper, we study *quiescent consistency* as proposed by Shavit and Herlihy, which is one such relaxed condition. More precisely, we give the first formal definition of quiescent consistency, investigate its relationship with linearizability, and provide a proof technique for it based on (coupled) simulations. We demonstrate our proof technique by verifying quiescent consistency of a (non-linearizable) FIFO queue built using a diffraction tree.

1 Introduction

The growth of multi- and many-core architectures has led to the increased use of algorithms that allow multiple processes to access and update a single shared data structure. Typically, these algorithms are concurrent (more efficient) re-implementations of standard data structures such as stacks, queues, sets, etc. Simple concurrent algorithms use locks to control access to the shared state, but more sophisticated algorithms dispense with locking and use non-blocking primitives such as compare-and-swap for synchronisation, enabling a finer granularity of atomicity. Because fine-grained atomicity increases the potential for parallelism, which in turn improves efficiency, such algorithms are set to become increasingly commonplace [19, 13].

The subtlety and complexity of fine-grained concurrent algorithms necessitates *formal verification* of their correctness. Several notions of correctness have been proposed including sequential consistency, quiescent consistency, and linearizability, which are defined by mapping the behaviours of a concurrent data structure to the behaviours of the corresponding abstract (sequential) data structure.

To date, most attention has been focused on linearizability as introduced by Herlihy and Wing [14], which requires that each operation call appears to take effect instantaneously at some point between its invocation and response. A number of approaches to proving linearizability have been developed, and several algorithms have been shown to be linearizable [5, 21, 3, 17, 9, 20]. The methodology used in these proofs varies, and ranges from shape analysis and separation logic to rely-guarantee reasoning and simulation-based methods.

However, linearizability is not the only relevant condition – weaker notions such as sequential consistency [15], quiescent consistency [13], and eventual consistency [18], as well as relaxed forms of linearizability like quasi linearizability [2] and *k*-linearizability [12] have also been defined. As algorithm designers seek to further decrease contention among the parallel processes (and increase efficiency) [19], these weaker criteria are set to become increasingly important. Below, we shall see example algorithms that use counting networks [4] and diffraction trees [1] to reduce contention. These algorithms are not linearizable and only satisfy the weaker correctness criteria.

In his recent paper [19], Shavit proposes quiescent consistency as a promising correctness condition for concurrent data structures in the multi-core age. Stated informally, quiescent consistency requires (1) operations to appear in a one-at-a-time sequential order, and (2) operations separated by a period of quiescence (i.e., a period in which no operation is executing) to appear to take effect in their real-time order. Hence, whenever an object becomes quiescent, its execution thus far must be equivalent to some sequential execution [13]. However, despite this simple formulation there appears to be no formal definition in the literature, let alone proof methodology to verify that an algorithm is quiescent consistent. This paper addresses this shortcoming, and our aim is to give the first formal definition of quiescent consistency and provide a proof technique for it based on coupled simulations.

Coupled simulations [11] are a proof methodology used in *refinement* - developed as an approach to non-atomic refinement in state-based systems [8], where the atomic abstract operations are implemented via a non-atomic decomposition. Refinement techniques have already proved useful in the verification of linearizability, see [10]. Here, we employ coupled simulations to derive a methodology for showing that fine-grained atomic concurrent algorithms are quiescent consistent, and apply it to prove quiescent consistency of a concurrent queue implementation. Moreover, the latter proof is fully mechanized using the interactive prover KIV [16].

The structure of this paper is as follows. In Section 2 we illustrate quiescent consistency via two versions of a concurrent queue implementation, and in Section 3 we give its formal definition. Background to the refinement and coupled simulation methodology is provided in Section 4, and this is applied to prove quiescent consistency of one of the queue versions in Section 5. Finally we conclude in Section 6.

2 Background

Quiescent consistency is a consistency requirement on concurrent data structures that is weaker than linearizability, and therefore allows more optimizations via reduced contention on the shared variables. In this section we present two queue implementations: a non-blocking and a blocking queue, each of which is based on the architecture of *diffracting trees*. We use these examples to illustrate linearizability and quiescent consistency and the difference between these conditions.

The architecture of diffracting trees uses the following principle (adapted from counting networks [4]): elements called *balancers* are arranged in a binary tree (with an arbitrary depth). Each balancer contains one bit, which determines the direction in which the tree is traversed; a balancer value of 0 causes a traversal up and a value 1 causes a traversal down. The leaves of the tree point to a concurrent data structure. Operations on the tree (and hence data structures) start at the root of the tree and traverse the tree based on the balancer values. Each traversal is coupled with a bit flip, so that the next traversal occurs along the other branch. Upon reaching a leaf, the process performs a corresponding operation on the data structure at the leaf.

The running example used for the rest of this paper is an implementation of a queue made up of two balancers and two queues, i.e., two diffracting trees with just one level, one for *dequeue* and one for *enqueue* operations (see Figure 1). Enqueue and dequeues share the two queues at the leaves of the trees.

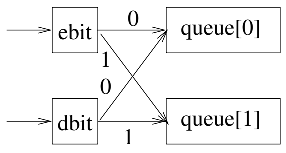


Fig. 1. A queue composed of two diffraction trees of level 1 and two queues

The two operations *enqueue* and *dequeue* are implemented as follows (where *Enq* and *Deq* are used to denote an atomic enqueue and dequeue):

```
    enqueue(el:T)                      dequeue
E1:    do lbit:=ebit;              D1:    do lbit:=dbit;
E2:    until                       D2:    until
          CAS(ebit,lbit,1-lbit)              CAS(dbit,lbit,1-lbit)
E3:    Enq(queue[lbit],el)         D3:    return Deq(queue[lbit])
```

Here, the semantics of CAS (Compare-And-Swap) is that of an atomic comparison of the stored local value with the shared variable followed by an assignment to the variable if the values are still equal:

```
CAS(var,old,new) = atomic{if var=old then var:=new;return true
                                     else return false}
```

In the implementation both operations read their corresponding bit and try to flip it. When they succeed, they *enqueue* (or *dequeue*) the queue of their local bit. The two queues work in FIFO order. There are two versions of the dequeue operation: a *non-blocking* version, which returns *empty* when the *Deq*ueue operation is executed on an empty queue, and a *blocking* version, where the *Deq* waits until an element is found in the queue. We will see that the former is not quiescent consistent while the latter is.

Now we take a look at the correctness conditions. We expect this structure to behave like a queue, i.e., operate in FIFO order and, of course, never return a value by a *dequeue* which has not been *enqueue*d before. Consistency conditions for concurrent data structures capture such expectations.

The general set up is as follows. Consistency requirements are usually defined via a comparison of the *histories* of concurrent implementations and an atomic abstract specification of the data structure. Histories are sequences of *events*, which can be invocations and returns of particular operations (out of some set I) by particular processes from a set P. Thus we define:

$$Event ::= inv\langle\langle P \times I \times IN \rangle\rangle \mid ret\langle\langle P \times I \times OUT \rangle\rangle$$

Here, IN and OUT are the domains for inputs and outputs (which include a null element), respectively. Operation calls by concurrent processes may overlap, but those by a single process are sequential. An operation call is *pending* if it has been invoked but has not yet returned. An object (data structure) is *quiescent* if it has no pending operation calls. Two operation calls are *ordered* if the return of the first operation call precedes the invocation of the second.

Example 1. If we let $P = \mathbb{N}, I = \{enq, deq\}$ and $IN = OUT = \{a, b, c, \ldots\}$, a possible history for the blocking concurrent queue implementation is the following:

$$\begin{aligned} h_1 \mathrel{\widehat{=}} \langle & inv(1, deq,), inv(2, enq, a), ret(2, enq,), inv(3, enq, b), ret(3, enq,), \\ & inv(4, deq,), ret(4, deq, b), inv(5, deq,), ret(5, deq, a), \\ & inv(6, enq, c), ret(6, enq,), ret(1, deq, c)\rangle \end{aligned}$$

There is not much concurrency in this run: only the first dequeue is running concurrently with the rest of the operations. Once started this dequeue is always pending until the end of the history, so the history is quiescent initially and at the end only. Note that the first dequeue must have already flipped the *dbit* when it starts. Thus the second dequeue returns the element in the lower queue which is b. □

The essential question of correctness is then to ask: Is this history a correct queue behaviour? Two different ways of answering it are the following (first given informally).

Linearizability: Operation calls should appear to take effect in their order.

Quiescent consistency: Operation calls separated by a period of quiescence should appear to take effect in their order.

Thus, linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its return. For quiescent consistency this requirement is relaxed. A "meaningful" explanation of a history must only be defined when the concurrent data structure in question is quiescent.

For example, history h_1 is not linearizable: the first two enqueues start and finish before the second and third dequeue, yet the dequeues return the elements in the reverse order. This is because the first dequeue has already flipped the *dbit*. The linearizability criterion therefore cannot be met. However, it turns out that the blocking implementation, and in particular h_1, is quiescent consistent: none of the intermediate states of h_1 are quiescent, and thus the consistency condition is not imposing any constraints on orderings. So we can indeed find an appropriate sequential history which has the same outcome as h_1, namely for instance the following one:

$$\begin{aligned}h_2 = \langle & \mathit{inv}(3, \mathit{enq}, b), \mathit{ret}(3, \mathit{enq},), \mathit{inv}(2, \mathit{enq}, a), \mathit{ret}(2, \mathit{enq},),\\ & \mathit{inv}(4, \mathit{deq},), \mathit{ret}(4, \mathit{deq}, b), \mathit{inv}(5, \mathit{deq},), \mathit{ret}(5, \mathit{deq}, a),\\ & \mathit{inv}(6, \mathit{enq}, c), \mathit{ret}(6, \mathit{enq},), \mathit{inv}(1, \mathit{deq},), \mathit{ret}(1, \mathit{deq}, c)\rangle\end{aligned}$$

Formally, one uses a *matching function* to relate each concurrent history (e.g., h_1) to a consistent sequential history (e.g., h_2) that "explains" the behaviour of the concurrent data structure with respect to a sequential execution. The requirements on the matching function are dependent on the consistency condition under consideration.

The non-blocking version of the algorithm (in which dequeue returns *empty* on empty queues) is not even quiescent consistent. This can be seen by the following history of the non-blocking queue:

$$\begin{aligned}h_3 \mathrel{\widehat=} \langle & \mathit{inv}(1, \mathit{deq},), \mathit{ret}(1, \mathit{deq}, \mathit{empty}), \mathit{inv}(2, \mathit{enq}, a), \mathit{ret}(2, \mathit{enq}),\\ & \qquad\qquad\qquad \mathit{inv}(3, \mathit{deq},), \mathit{ret}(3, \mathit{deq}, \mathit{empty})\rangle\end{aligned}$$

In h_3 we find lots of quiescent states, which necessitate keeping the order of operations. The second dequeue does, however, not return the a which – due to the prior enqueue – should be the result. Nevertheless, the blocking version of the queue *is* quiescent consistent, and in the following we will precisely define what this means and how we can prove it.

3 Quiescent consistency

In this section, we formalise both linearizability and quiescent consistency with their informal definitions in mind. Both notions of consistency compare a (possibly highly concurrent) implementation with an abstract sequential specification S. In S, operations (like *enqueue* and *dequeue*) are executed atomically. The consistency conditions then compare the histories of the implementation and specification, and reorder the implementation's histories in some way so that it matches the specification. Each consistency condition formalises the allowed reorderings within the histories.

First of all, not all sequences of events are correct histories. Thus we need the notion of a *legal* history: one that consists of matching pairs of invoke and return events plus possibly some pending invocations, where an operation has started but not yet finished.

To formalise this we need some notation. We let *History* denote the set of all histories. For a history h, $\#h$ is the length of the sequence, and $h(n)$ its nth element (for $n : 1..\#h$). We use predicates $\mathit{inv}?(e)$ and $\mathit{ret}?(e)$ to check whether an event $e \in \mathit{Event}$ is an invoke or a return, and we let $\mathit{Ret}!$ be the set of return events. We let $e.p \in P$ be the process executing the event e and $e.i \in I$ the index of the abstract operation to which the event belongs. We can then define a legal history:

Definition 1. *Let $h : \mathit{seq}\,\mathit{Event}$ be a sequence of events. Two positions m, n in h form a* matching pair*, denoted $mp(m, n, h)$ if*

$$\begin{aligned}& 0 < m < n \leq \#h \land h(m).p = h(n).p \land h(m).i = h(n).i \land\\ & \forall k \bullet m < k < n \Rightarrow h(k).p \neq h(m).p\end{aligned}$$

A position n in h is a pending invocation, *denoted* $pi(n,h)$, *if*

$$1 \leq n \leq \#h \wedge inv?(h(n)) \wedge \forall m \bullet n < m \leq \#h \Rightarrow h(m).p \neq h(n).p$$

h is legal, *denoted* $legal(h)$, *if*

$$\begin{array}{l} \forall n : 1..\#h \bullet \textbf{if } inv?(h(n)) \textbf{ then } pi(n,h) \vee \exists m : 1..\#h \bullet mp(n,m,h) \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \exists m : 1..\#h \bullet mp(m,n,h) \end{array} \qquad \Box$$

A history is *sequential* if all invoke operations are immediately followed by their matching returns. In the examples above, history h_1 is not sequential, whereas h_2 and h_3 are, and all are legal. Having defined the notion of pending invocation, we can now fix what we mean by a quiescent state, or more precisely, quiescent history.

Definition 2. *A legal history h is* quiescent, *written* qu(h), *if* $\neg\exists n \bullet pi(n,h)$. $\Box$

Both the definition of linearizability and quiescent consistency are given by comparing the histories generated by concurrent implementations with the sequential histories of some given abstract atomic specification. For the moment, we just assume legal histories to be given; in the next section we will precisely define these for our queue.

Definition 3 (Quiescent consistency). *Let h be a quiescent, concurrent history, hs a sequential history. The history h is said to be* quiescent consistent *with hs, denoted* $qcons(h,hs)$, *if*

$$\begin{array}{l} \exists \textit{ bijective } f : 1..\#h \rightarrowtail\!\!\!\!\twoheadrightarrow 1..\#hs \bullet \\ \qquad (\forall n : 1..\#h : h(n) = hs(f(n))) \wedge (\forall m,n : mp(m,n,h) \Rightarrow f(m)+1 = f(n)) \\ \quad \wedge \forall m,n,k : m < n \wedge m \leq k \leq n \wedge qu(h[1..k]) \wedge ret?(h(m)) \wedge inv?(h(n)) \\ \qquad\quad \Rightarrow f(m) < f(n) \end{array}$$

An implementation I is quiescent consistent *wrt. a specification S if for all quiescent histories h of I there is a sequential history hs of S such that* $qcons(h,hs)$. $\Box$

Our definition allows operations of a quiescent history *h* (represented as matching pairs) to be reordered arbitrarily between quiescent states. However, each individual matching pair must be preserved according to the second conjunct of the definition of *qcons*.

For quiescent consistency we look at quiescent histories. Linearizability considers all histories of the implementation and first brings each non-quiescent history into a ”reasonable” quiescent one. To this end, it extends the history with the return events of those operations which ”have taken effect”, and afterwards it removes the remaining pending invokes using a function *complete*.

Definition 4 (Linearizability). *Let h be a history, hs a sequential history. The history h is said to be in lin-relation with hs, denoted* $lin(h,hs)$, *if*

$$\begin{array}{l} \exists \textit{ bijective } f : 1..\#h \rightarrowtail\!\!\!\!\twoheadrightarrow 1..\#hs \bullet \\ \qquad (\forall n : 1..\#h \bullet h(n) = hs(f(n))) \wedge (\forall m,n : mp(m,n,h) \Rightarrow f(m)+1 = f(n)) \\ \quad \wedge \forall m,n,m',n' : 1..\#h \bullet n < m' \wedge mp(m,n,h) \wedge mp(m',n',h) \Rightarrow f(n) < f(m') \end{array}$$

A concurrent history h is linearizable *with respect to some sequential history hs, denoted* $linearizable(h,hs)$, *if*

$$\exists h_0 : seq\,Ret! \bullet legal(h \frown h_0) \wedge lin(complete(h \frown h_0), hs)$$

An implementation I is linearizable *with a specification S if for all histories h of I there is a sequential history hs of S such that linearizable*(h, hs). □

It is easy to see that linearizability is the stronger notion.

Proposition 1. *Let h be a quiescent, hs a sequential history. Then*
$lin(h, hs) \Rightarrow qcons(h, hs)$ □

4 Coupled simulations - a proof methodology

Before presenting a proof technique for quiescent consistency, we need to fix the implementation and abstract specification. Both are given as abstract data types of the form: $S = (State, Init, (Op_{p,i})_{p \in P, i \in I})$, consisting of a state, an initialisation condition, a collection of operations. (Because each process can execute each operation, they are indexed by the process id.) As we will use techniques from the area of refinement, the abstract sequential specification will be called A (abstract) while the implementation is the concrete level and thus named C. We formalise the data types within Z.

$$\begin{array}{|l}\hline AState \\ queueA : \mathrm{seq}\, T \\ \hline \end{array} \qquad \begin{array}{|l}\hline AInit \\ AState' \\ \hline queueA' = \langle\, \rangle \\ \hline \end{array}$$

$$\begin{array}{|l}\hline AEnq_p \\ \Delta AState \\ el? : T \\ \hline queueA' = queueA \frown \langle el? \rangle \\ \hline \end{array} \qquad \begin{array}{|l}\hline ADeq_p \\ \Delta AState \\ el! : T \\ \hline queueA = \langle el! \rangle \frown queueA' \\ \hline \end{array}$$

Note that this specifies a blocking queue: the dequeue operation can only be executed if the queue can be divided into one element and the rest.

The implementation based on diffraction trees needs a bit more explanation. In general, we need to distinguish in all such concurrent data structures the global state (here, the two balancers and the two queues) and the local variables of the processes (here, the input parameter *el* for the enqueue, the local bit *lbit* plus a program counter). Recall that P is the set of all process identifiers. For the program counters, the values will be the line numbers plus one value N standing for a process being idle: $PC = \{N, E1, E2, E3, D1, D2, D3\}$.

$$\begin{array}{|l}\hline CState \\ ebit : \mathbb{B}, dbit : \mathbb{B} \\ queueC : \mathbb{B} \to \mathrm{seq}\, T \\ lbit : P \to \mathbb{B} \\ el : P \to T \\ pc : P \to PC \\ \hline \end{array} \qquad \begin{array}{|l}\hline CInit \\ CState' \\ \hline ebit' = dbit' = 0 \\ queueC'(0) = \langle\, \rangle \\ queueC'(1) = \langle\, \rangle \\ \forall p \in P \bullet pc'(p) = N \\ \hline \end{array}$$

For every line in the algorithm and every process possibly executing it, we now define one operation in the concrete implementation data type. We refrain from giving all of them here, and just give two examples. We use the Object-Z convention that all variables which are not named in the schema remain the same.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{enq2C_p}}} \\
\Delta CState \\
\hline
pc(p) = E2 \\
lbit(p) = ebit \Rightarrow \\
\quad ebit' = 1 - lbit(p) \wedge pc'(p) = E3 \\
lbit(p) \neq ebit \Rightarrow \\
\quad ebit' = ebit \wedge pc'(p) = E1 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{deq3C_p}}} \\
\Delta CState \\
el! : T \\
\hline
pc(p) = D3 \\
queueC(lbit(p)) = \\
\qquad \langle el! \rangle \frown queue'(lbit(p)) \\
pc'(p) = N \\
\hline
\end{array}$$

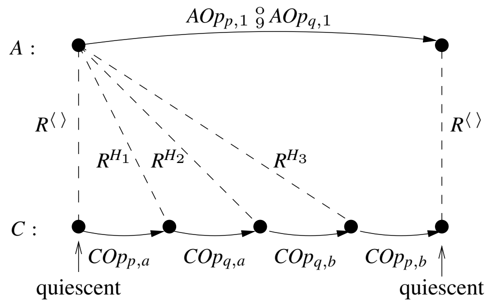
The basic idea for a proof method for quiescent consistency is to compare abstract specification and concrete implementation data type with respect to some notion of *refinement* [7]. The standard proof strategy for refinement proceeds via simulations which come in two forms (which are sound and jointly complete), forward and backward simulation [6]. Here, we first of all aim at a sound proof technique for quiescent consistency and thus use just one, namely forward simulation (a complete technique would probably need backward simulations as well). We furthermore can elide the condition of applicability since quiescent consistency is a safety property rather than a general refinement property where one would need to ensure progress of the concrete system.

Definition 5 (Forward simulation).
Let $A = (AState, AInit, (AOp_{p,i})_{p \in P, i \in I})$ *and* $C = (CState, CInit, (COp_{p,i})_{p \in P, i \in I})$ *be two data types. A relation* $R : AState \times CState$ *is a* forward simulation *from A to C if the following two conditions hold:*

- *Initialization:* $\forall ci : CInit \bullet \exists ai : AInit \bullet R(ai, ci)$,
- *Correctness:*
  $\forall as : AState, cs : CState, cs' : CState, in : IN, out : OUT, p : P, i : I \bullet$
  $\qquad R(as, cs) \wedge COp_{p,i}(in, cs, cs', out) \Rightarrow$
  $\qquad (\exists as' : AState \bullet R(as', cs') \wedge AOp_{p,i}(in, as, as', out))$ □

The two conditions state that (a) every initial concrete state needs to have a matching (via the relation R - known as a retrieve relation) initial abstract state and (b) all the steps of the concrete data type need to be matched by corresponding abstract steps. Here, the assumption is that the granularity of data types is the same: every concrete operation has exactly one corresponding abstract operation. This assumption needs to be relaxed for our application; in fact for all applications which carry out some sort of non-atomic refinement where an abstract operations is implemented by a whole sequence of concrete operations. Thus, we assume the operations of the abstract data type A are indexed by (process names plus) elements from some set I, and operations of C indexed by elements from some set J (plus again process names), and an abstraction function $abs : J \to I$ is given.

For the queue, all concrete enq_p's are related to $AEnq_p$ and similar for dequeue. For a non-atomic refinement, we furthermore need to know what the operations are which start (invoke) an implementation sequence, which end (return from an invocation of) a sequence and which are internal. For the former two we use the predicates *inv*? and *ret*? defined in the last section, for the latter we use a similar predicate *int*?.



$$H_1 = \langle inv(p, AOp_1,)\rangle$$
$$H_2 = H_1 \frown \langle inv(q, AOp_1,)\rangle$$
$$H_3 = H_2 \frown \langle ret(q, AOp_1,)\rangle$$

Fig. 2. Coupled simulation for some example run

The basic idea of the non-atomic, or coupled, simulation which we use in the following, is to match only the return steps of a sequence with the abstract operations, and (abstractly) view all other steps as "skip" steps. In addition, the matching of return steps only takes place when we have arrived at a quiescent consistent history again. To keep track of the progress of the concrete operation, we extend the retrieve relation R with histories H, thus getting a family of retrieve relations R^H. When we finally reach a quiescent history (by executing a return operation), we need to match up with *all* abstract operations occuring in H. However, quiescent consistency allows us to look for just some sequential order, not necessarily in the order of them appearing in H.

Figure 2 shows an example of this where the abstract operation $AOp_{p,1}$ is implemented as $COp_{p,a} \mathbin{;} COp_{p,b}$, i.e., $abs : a \mapsto 1, b \mapsto 1$. The diagram shows some steps of the concrete system in which processes p and q are running (i.e., $COp_{p,a}$ is the execution of operation COp_a by process p and so on), and how this would be simulated in the abstract. Only when we reach a quiescent state again, we need to match up with the abstract. During non-quiescent concrete states the retrieve relation R^H relates the concrete states to the "previous" abstract state.

We are now going to formally define this type of simulation. We write $h \simeq hs$ for two histories h, hs iff they are permutation equivalent and matching pairs are preserved. We let AOP denote the set of all abstract operations. For a sequential history hs and abstract states as, as' we define

$$hs(as, as') \mathrel{\widehat{=}} \exists aops : AOP^* \bullet aops(as, as') \wedge hist(aops) = hs,$$

where *hist* makes a proper history out of a sequence of abstract operations,

$$hist(\langle AOp_{p_1,1}(in_1, out_1), \ldots, AOp_{p_n,n}(in_n, out_n)\rangle) = \\ \langle inv(p_1, AOp_1, in_1), ret(p_1, AOp_1, out_1), \ldots, inv(p_n, AOp_n, in_n), ret(p_n, AOp_n, out_n)\rangle.$$

Definition 6 (Coupled simulation). *Let* $A = (AState, AInit, (AOp_{p,i})_{p\in P, i\in I})$ *be an abstract data type and* $C = (CState, CInit, (COp_{p,j})_{p\in P, j\in J})$ *a concrete data type, related* *via abstraction function* $abs : J \to I$*. A family of relations* $R^H \subseteq AState \times CState$*,* H *a history of* C*, is a* coupled simulation relation *from* A *to* C *if the following holds:*

- *Initialization:* $\forall ci : CInit \bullet \exists ai : AInit \bullet R^{\langle\rangle}(ai, ci)$,
- *Correctness:*
  1. *Invocation:* $\forall as : AState, cs : CState, cs' : CState, in : IN, p : P, j : J \bullet$
     $R^H(as, cs) \wedge COp_{p,j}(in, cs, cs') \wedge inv?(COp_{p,j}) \Rightarrow R^{H ^\frown \langle inv(p, AOp_{abs(j)}, in)\rangle}(as, cs')$,
  2. *Internal:* $\forall as : AState, cs : CState, cs' : CState, p : P, j : J \bullet$
     $R^H(as, cs) \wedge COp_{p,j}(cs, cs') \wedge int?(COp_{p,j}) \Rightarrow R^H(as, cs')$,
  3. *Return to quiescent:*
     $\forall as : AState, cs : CState, cs' : CState, out : OUT, p : P, j : J \bullet$
     $R^H(as, cs) \wedge COp_{p,j}(cs, cs', out) \wedge ret?(COp_{p,j}) \wedge qu(H ^\frown \langle ret(p, AOp_{abs(j)}, out)\rangle) \Rightarrow$
     $\exists as' : AState \bullet R^{\langle\rangle}(as', cs') \wedge$
     $\exists$ *sequential* $hs \bullet hs \simeq H ^\frown \langle ret(p, AOp_{abs(j)}, out)\rangle \wedge hs(as, as')$,
  4. *Return to non-quiescent:*
     $\forall as : AState, cs : CState, cs' : CState, out : OUT, p : P, j : J \bullet$
     $R^H(as, cs) \wedge COp_{p,j}(cs, cs', out) \wedge ret?(COp_{p,j}) \wedge \neg qu(H ^\frown \langle ret(p, AOp_{abs(j)}, out)\rangle)$
     $\Rightarrow R^{H ^\frown \langle ret(p, AOp_{abs(j)}, out)\rangle}(as, cs')$. □

It can be shown that coupled simulation is a sound proof technique for quiescent consistency (the proof of this follows easily from the definition.):

Theorem 1. *Let* $A = (AState, AInit, (AOp_{p,i})_{p \in P, i \in I})$ *be an abstract data type and* $C = (CState, CInit, (COp_{p,j})_{p \in P, j \in J})$ *a concrete data type. If there is a coupled simulation* R^H *from* A *to* C*, then* C *is quiescent consistent wrt.* A*.* □

Moreover, the two simulation types – forward and coupled simulation – can safely be combined.

Proposition 2. *For some abstract data types* $A = (AState, AInit, (AOp_{p,i})_{p \in P, i \in I})$*,* $B = (BState, BInit, (BOp_{p,j})_{p \in P, j \in J})$ *and* $C = (CState, CInit, (COp_{p,j})_{p \in P, j \in J})$ *related via* $abs : J \to I$*. If there is a coupled simulation* R^H *from* A *to* B *and a forward simulation* S *from* B *to* C*, then we have a coupled simulation relation from* A *to* C*.*

Proof: Define a coupled simulation from A to C by $S^H = R^H \fatsemi S$. □

In the next section, we will make use of coupled simulations and their combination with forward simulations to show quiescent consistency of the blocking queue.

5 Quiescent consistency of the blocking queue

To prove quiescent consistency of the blocking queue implementation, we proceed in two steps. Instead of directly constructing a coupled simulation relation between A and C, we introduce an intermediary data type (called B), and then show the existence of a coupled simulation between A and B, and a (trivial) forward simulation from B to C. The coupled simulation proofs have been mechanized with the interactive prover KIV.[1] The abstract data type B includes all of C's state plus some auxiliary information to help us in the proof. First, it records the values of $queueC(i)$, $ebit$ and $dbit$ from the last quiescent state as $lastq(i)$, $lastEbit$ and $lastDbit$ (for $i = 0, 1$). It also records the processes which have done enqueues and dequeues since then in $enqs(i)$ and $deqs(i)$. Two auxiliary queues $auxq(i)$ store $lastq(i)$ plus all the enqueued elements since the last quiescent state. Dequeued elements are not removed from $auxq(i)$.

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ BState\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
CState \\
lastq, auxq : \mathbb{B} \rightarrow \operatorname{seq} T \\
enqs, deqs : \mathbb{B} \rightarrow \operatorname{seq} P \\
lastEbit, lastDbit : \mathbb{B} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\_\ BInit\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
BState' \\
\hline
CInit \\
lastEbit' = lastDbit' = 0 \\
\forall i : \{0,1\} \bullet auxq'(i) = lastq'(i) = \langle\,\rangle \\
\qquad\qquad \wedge\ enqs'(i) = deqs'(i) = \langle\,\rangle \\
\hline
\end{array}$$

All operations on C are extended to operations on state B. For all but the last operations $deq3C_p$ and $enq3C_p$ of each algorithm the extension leaves the auxiliary state unchanged. Formally, e.g., $deq2B_p = deq2C_p \wedge \Xi(BState \setminus CState)$, where $\Xi(S)$ denotes the identity relation on S.

Operations $deq3C_p$ and $enq3C_p$ get extended twice. First they must modify $auxq$, $enqs$ and $deqs$ appropriately. For *enqueue*, the new element is appended to the auxiliary queue and the process id to the sequence of enqueues (operation $weakenq3B_p$). Operation $weakdeq3B_p$ is similar, except that dequeues are not applied to $auxq$.

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ weakenq3B_p\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta BState \\
\hline
enq3C_p \\
auxq'(lbit(p)) = auxq(lbit(p)) \frown el(p) \\
enqs'(lbit(p)) = enqs(lbit(p)) \frown \langle p \rangle \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\_\ weakdeq3B_p\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta BState \\
el! : T \\
\hline
deq3C_p \\
deqs'(lbit(p)) = deqs(lbit(p)) \frown \langle p \rangle \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ resetB\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta BState \\
\hline
\Xi CState \\
\mathbf{if}\ \forall p : P \bullet pc(p) = N \\
\mathbf{then}\ auxq' = lastq' = queueC \wedge lastEbit' = ebit \wedge lastDbit' = dbit \\
\mathbf{else}\ \Xi(BState \setminus CState) \\
\hline
\end{array}$$

Furthermore, when the step brings B into a quiescent state, we have to appropriately reset the auxiliary information. This is done by sequentially composing with *resetB*, i.e.

$$enq3B_p = weakenq3B_p \,⨾\, resetB \qquad deq3B_p = weakdeq3B_p \,⨾\, resetB$$

[1] See https://swt.informatik.uni-augsburg.de/swt/projects/QC-queue.html for a description of the KIV proofs.

A number of invariants are valid for the reachable part of this data type, for instance $\forall i \in \{0,1\} \bullet \#enqs(i) \leq \#auxq(i) \wedge \#deqs(i) \leq \#auxq(i)$. The queues are related in the following way:

$$lastq(i) = auxq(i)[1..(\#auxq(i) - \#enqs(i))]$$
$$queueC(i) = auxq(i)[(\#deqs(i)+1)..\#auxq(i)]$$

For the proof of coupled simulation, we need one rather important invariant stating a connection between the sizes of the two queues, the number of already enqueued and dequeued elements, the number of pending enqueues and dequeues and the two bits. For this, we define pending enqueues and dequeues to be the following.

$$PE(i) = \{p \in P \mid pc(p) = E3 \wedge lbit(p) = i\}$$
$$PD(i) = \{p \in P \mid pc(p) = D3 \wedge lbit(p) = i\}$$

The invariant states a balancing property:

Proposition 3. *Let bs : BState a reachable state of the abstract data type B. Then the following invariant* **INV** *holds:*

$$\#queueC(0) + \#PE(0) - \#PD(0) + dbit =$$
$$\#queueC(1) + \#PE(1) - \#PD(1) + ebit\ .$$

Proof sketch: By induction on the number of steps needed to reach the state. Initially, the invariant holds as all sequences are empty and *dbit* and *ebit* are both 0. For the induction step there are a number of cases to consider. As one example: assume the next operation is from process p, moving from $E2$ to $E3$ thereby increasing the size of $PE(lbit(p))$ by one. If $lbit(p) = 0$ (enqueue to upper queue), this furthermore sets *ebit* from 0 to 1 thereby keeping the sums of both sides equal. If on the other hand $lbit(p) = 1$ (enqueue going to lower queue), *ebit* is set from 1 to 0 thereby keeping the sum on the right hand side of the equation the same. □

Note that for quiescent states the equation reduces to $\#queueC(0) + dbit = \#\ queueC(1) + ebit$, i.e., the two queues are balanced: in quiescent states they can differ in size by at most one, and the two bits specify the allowed difference.

Next we go to the central part of our proof, the abstraction relation for the coupled simulation R^H. It should relate states of A and B with particular histories H. First of all, we consider the case when H is empty, i.e., and in a quiescent state. In this case, we just need to determine the contents of the abstract queue from the two concrete queues $queueC(0)$ and $queueC(1)$ by shuffling their contents, starting with $queueC(dbit)$ (since this is where dequeueing processes start). As the above invariant **INV** tells us, in size the two queues can be just one element apart. Therefore the following recursive definition of shuffle (which leaves, e.g., ($shuffle(0, \langle\,\rangle, q)$ for nonempty q unspecified) is sufficient:

$$shuffle(0, \langle\,\rangle, \langle\,\rangle) = \langle\,\rangle \qquad shuffle(1, \langle\,\rangle, \langle\,\rangle) = \langle\,\rangle$$
$$shuffle(0, \langle a \rangle \frown q_1, q_2) = \langle a \rangle \frown shuffle(1, q_1, q_2)$$
$$shuffle(1, q_1, \langle a \rangle \frown q_2) = \langle a \rangle \frown shuffle(0, q_1, q_2)$$

For quiescent states $queueA = shuffle(dbit, queueC(0), queueC(1))$. For non-quiescent states, we need to link up with the abstract queue which was represented in the last quiescent state (see the simulation diagram in Figure 2). Thus, we then simply use $queueA = shuffle(lastDbit, lastq(0), lastq(1))$. In quiescent states, *lastDbit* and *dbit* as well as $queueC(i)$ and $lastq(i)$ coincide; thus the last expression is valid both for quiescent and non-quiescent states.

Now to the history H: It accumulates the invocation and return events which have happened since the last quiescent state. The order is in fact irrelevant as we seek to find a matching sequential history which is just a permutation ($\simeq$) of H. However, the events inside H have to be consistent with the auxiliary information of *BState*: for every process in $enqs(i)$ there has to be an invoke and a return event in H (plus similiar for processes in $deqs(i)$) and if there is a currently running *enqueue* (*dequeue*, respectively) there furthermore has to be an *invoke* event for it.

To formalize this, we construct a sequence of events from $enqs(i)$ and $deqs(i)$. The necessary information about enqueued / dequeued elements is found in $auxq(i)$:

$$\begin{aligned} evts(enqs(i)) = {\frown}_{j=1..\#enqs(i)} \langle & inv(p, enq, a), ret(p, enq,) \bullet \\ & p = enqs(i)(j) \wedge a = auxq(i)(\#lastq(i) + j)\rangle \\ evts(deqs(i)) = {\frown}_{j=1..\#deqs(i)} \langle & inv(p, deq,), ret(p, deq, a) \bullet \\ & p = deqs(i)(j) \wedge a = auxq(i)(j)\rangle \end{aligned}$$

Last, we let $invevts(bs)$ be the invoke events of currently running enqueues and dequeues in state bs (the order is irrelevant), i.e.,

$$\begin{aligned} invevts(bs) = & \langle inv(p, enq, a) \bullet pc(p) \in \{E1, E2, E3\} \wedge el(p) = a\rangle \frown \\ & \langle inv(p, deq,) \bullet pc(p) \in \{D1, D2, D3\}\rangle \end{aligned}$$

With these definitions at hand, we can state the second theorem.

Theorem 2. *Let A and B be the abstract data types defined above. Then*

$$\begin{aligned} R^H \mathrel{\widehat{=}}\ & queueA = shuffle(lastDbit, lastq(0), lastq(1)) \wedge \\ & H \simeq evts(enqs(0)) \frown evts(enqs(1)) \frown \\ & \quad evts(deqs(0)) \frown evts(deqs(0)) \frown invevts(bs) \end{aligned}$$

is a coupled simulation from A to B.

Proof: Since the abstraction function only changes when the resulting state is quiescent (when the positive case of *resetB* is executed), the critical proof obligation is the case "Return to quiescent" in Def. 6. It requires constructing a suitable sequential history *hs*. This history consists of two halves. The first half executes the enqueues of $shuffle(\neg ebit, evts(enqs(0), evts(enqs(1))$ in reverse order[2] resulting in the abstract queue $shuffle(lastDbit, auxq(0), auxq(1))$. The second part executes the dequeues of $shuffle(lastDbit, evts(deqs(0), evts(deqs(1))$ to get to the current abstract queue. The proof is inductive over the lengths of the *enq* and *deq* lists. □

This completes the part of the proof relating data types A and B. The second step is now the one from B to C for which we have a forward simulation.

[2] The KIV proof combines *evts*, shuffling and reversing into one function *eshuffle*.

Proposition 4. *Let B and C be the abstract data types defined above. Then there is a forward simulation from B to C.*

Proof: Directly follows from the fact that B is just an extension of C, or seen the other way round, C's operations being a projection of B's operations onto *CState*. □

This finally implies the correctness of the blocking queue implementation.

Corollary 1. *The data type C, i.e., the queue implementation with blocking dequeue operations, is quiescent consistent with respect to the abstract data type A.*

Proof: Follows from Theorems 1 and 2 together with Propositions 2 and 4. □

6 Conclusion

In this paper, we have given a formal definition of, and a proof methodology for, quiescent consistency. We have demonstrated the technique by proving quiescent consistency of a concurrent, non-linearizable queue implementation. To the best of our knowledge, this is the first formal proof of quiescent consistency of an algorithm.

We have chosen to formalise quiescent consistency in a way that matches the informal definition as closely as possible, however, since we are formalising an informal description there might be valid alternatives to our definition above. In particular, our formalisation has some specific consequences, since it embodies the idea that *quiescent consistency does not necessarily preserve program order*. This means that one is even allowed to reorder the operations of a single process. So the following history (not occuring for our queue example):

$\langle inv(1, enq, a), inv(2, deq,), ret(2, deq, b), inv(2, enq, b), ret(2, enq,), ret(1, enq,)\rangle$
where the b is visibly dequeued before being enqueued, is accepted, since it can be reordered to the sequential history:

$\langle inv(2, enq, b), ret(2, enq,), inv(2, deq,), ret(2, deq, b), inv(1, enq, a), ret(1, enq,)\rangle$.
However strange this may seem, it appears that most informal discussions on quiescent consistency view this as a consequence of the definition.

There are also other alternatives to quiescent consistency or linearizability for concurrent object correctness. For example, *eventual consistency* [18] states that all observations on a system will agree if there are no more updates to the system. Although this is a weaker condition than sequential consistency, there is no relation between it and quiescent consistency. In a similar way *quasi-linearizability* [2], or *k-linearizability* [12] and quiescent consistency are incomparable.

It is worth noting that although both quasi-linearizability and k-linearizability have been formally defined, neither condition has an associated proof method. Our aim here was to provide a proof method for quiescent consistency, which we did via the use of coupled simulations, and furthermore, show how these proofs can be mechanised. In particular we provided a full mechanisation of the coupled simulation proofs for the queue using KIV.

References

1. Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In P. D'Ambra, M. Rosario Guarracino, and D. Talia, editors, *Euro-Par (2)*, volume 6272 of *LNCS*, pages 151–162. Springer, 2010.
2. Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *OPODIS*, volume 6490 of *LNCS*, pages 395–410. Springer, 2010.
3. D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, pages 477–490, 2007.
4. J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, September 1994.
5. R. Colvin, S. Doherty, and L. Groves. Verifying concurrent data structures by simulation. *ENTCS*, 137:93–110, 2005.
6. W. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
7. J. Derrick and E. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Springer, May 2001.
8. J. Derrick and E. A. Boiten. Non-atomic refinement in Z. In J. M. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods*, volume 1709 of *LNCS*, pages 1477–1496. Springer, 1999.
9. J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanizing a correctness proof for a lock-free concurrent stack. In *FMOODS 2008*, volume 5051 of *LNCS*, pages 78–95. Springer, 2008.
10. J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4, 2011.
11. J. Derrick and H. Wehrheim. Using coupled simulations in non-atomic refinement. In D. Bert, J. P. Bowen, S. King, and M. A. Waldén, editors, *ZB*, volume 2651 of *LNCS*, pages 127–147. Springer, 2003.
12. T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. In *POPL*, pages 317–328. ACM, 2013.
13. M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.
14. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.
15. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.
16. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In *Automated Deduction—A Basis for Applications*, volume II, chapter 1: Interactive Theorem Proving, pages 13 – 39. Kluwer, 1998.
17. G. Schellhorn, H. Wehrheim, and J. Derrick. How to prove algorithms linearisable. In P. Madhusudan and S.A. Seshia, editors, *CAV 2012, USA, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 2012.
18. M. Shapiro and B. Kemme. Eventual consistency. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 1071–1072. Springer US, 2009.
19. N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.
20. B. Tofan, G. Schellhorn, and W. Reif. Formal verification of a lock-free stack with hazard pointers. In *Proc. ICTAC*, pages 239–255. Springer LNCS 6916, 2011.
21. V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP '06*, pages 129–136. ACM, 2006.