https://doi.org/10.1007/s00165-021-00541-8

Formal Aspects of Computing

Formal Aspects of Computing

Verifying correctness of persistent concurrent data structures: a sound and complete method

John Derrick[1], Simon Doherty[1], Brijesh Dongol[2], Gerhard Schellhorn[3], and Heike Wehrheim[4]

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

Abstract. Non-volatile memory (NVM), aka persistent memory, is a new memory paradigm that preserves its contents even after power loss. The expected ubiquity of NVM has stimulated interest in the design of *persistent* concurrent data structures, together with associated notions of correctness. In this paper, we present a formal proof technique for *durable linearizability*, which is a correctness criterion that extends linearizability to handle crashes and recovery in the context of NVM. Our proofs are based on refinement of Input/Output automata (IOA) representations of concurrent data structures. To this end, we develop a generic procedure for transforming any standard sequential data structure into a durable specification and prove that this transformation is both sound and complete. Since the durable specification only exhibits durably linearizable behaviours, it serves as the abstract specification in our refinement proof. We exemplify our technique on a recently proposed persistent memory queue that builds on Michael and Scott's lock-free queue. To support the proofs, we describe an automated translation procedure from code to IOA and a thread-local proof technique for verifying correctness of invariants.

Keywords: Non-volatile memory, Concurrent data structures, Refinement, Linearizability.

1. Introduction

Recent technological advances indicate that future architectures will employ some form of *non-volatile memory* (NVM) that retains its contents after a system crash (e.g., power outage). NVM is intended to be used as an intermediate layer between traditional *volatile memory* (VM) and secondary storage and has the potential to vastly improve system speed and stability. Software that uses NVM has the potential to be more robust; in case of a crash, a system state before the crash may be recovered using contents from NVM, as opposed to being restarted from secondary storage. However, because the same data is stored in both a volatile and non-volatile manner, and because NVM is updated at a slower rate than VM, recovery to a consistent state may not always be possible. This is particularly true for concurrent systems, where coping with NVM requires introduction of additional synchronisation instructions into a program.

Correspondence to: Brijesh Dongol, E-mail: b.dongol@surrey.ac.uk

Published online: 17 May 2021

Recently, researchers have developed *persistent* extensions to existing concurrent objects (e.g., concurrent data structures or transactional memory). This work has been accompanied by extensions to known notions of consistency, such as linearizability and opacity that cope with crashes and subsequent recovery.

This paper examines correctness and formal verification of the recently developed persistent queue by Friedman et al. [FHMP18], against the (also) recently developed notion of *durable linearizability* [IMS16]. Friedman et al.'s queue extends the well-known Michael-Scott queue [MS96], whereas durable linearizability extends the standard notion of linearizability [HW90] so that completed operations are guaranteed to survive a system crash.

Our verification follows a well-established methodology: (1) we develop an operational model of durable linearizability that is parameterised by a generic sequential object (e.g., a queue data structure with enqueue and dequeue operations), (2) we prove that this operational model both sound and complete, and (3) we establish a series of refinements between the operational model and the concrete implementation. The final (and most complex) of these steps, which establishes that the implementation refines the operational model, is fully mechanised in the KIV theorem prover [EPS+15]. It is important to note that the operational model is generic and for any particular verification one only needs to establish step (3) in order to show that a particular algorithm is durable linearizable.

Ours is the first approach to address formal verification of persistent data structures. We consider the development of our sound operational characterisation of durable linearizability and the refinement proofs, including mechanisation in KIV, to be the main contributions of this paper. The mechanisation may be accessed from [KIV20].

We present Friedman et al.'s queue in Section 2, durable linearizability in Section 3 and an operational characterisation of durable linearizability via Input/Output automata (IOA) in Section 4. Section 5 then gives an overview of our proof approach. It requires the translation of programs as given in a programming language into IOA which we describe further in Section 6. For proving refinement, we need to show invariants on the data structure which we carry out by thread-local proof techniques detailed in Section 7. Finally, Section 8 explains how we apply these generic proof concepts on the example of the persistent queue. This article is an extended version of [DDD+19] adding the technique for translating code to IOA and providing a compositional technique for invariant verification which our refinement approach heavily relies on.

2. A persistent queue

The persistent queue of Friedman et al. [FHMP18] is an extension of the Michael-Scott queue (MSQ) [MS96] to cope with NVM (see Algorithms 1 and 2). The MSQ uses a linked list of nodes with global head and tail pointers. The first node is a sentinel that simplifies handling of empty queues. The MSQ is initialised by allocating a dummy node with a null next pointer, then setting the global head and tail pointers to this dummy node.

The enqueue operation creates a new node that is inserted at the end of the linked list. The insertion is performed using an atomic compare-and-swap (CAS) instruction that atomically updates the *next* pointer of the last node provided this next pointer hasn't changed since it was read at the beginning of the enqueue operation. The CAS returns true if it succeeds and false otherwise. Immediately after a new node is inserted, the tail pointer is *lagging* one node behind the true tail of the queue, and hence, must be updated to point to the last node in a separate step.

The dequeue operation returns `empty` if the head and tail pointer both point to the sentinel node and the tail is not lagging. If the queue is not empty, the dequeue reads from the value of the node immediately after the sentinel and atomically swings the head pointer to this next node provided it has not changed. Thereby, the next node becomes the new sentinel node of the queue.

A key feature of MSQ is a *helping mechanism* where a different thread from the original enqueue may advance the tail pointer if it is lagging. In the case of a dequeue, this only occurs if head and tail pointers are equal, but the queue is not empty.

Algorithm 1 Constructors

```
1: class Node
2: T val;
3: Node* next;
4: int deqID;
5: Node(T k):
6:   val(k), deqID(-1), next(null);
```

```
1: class DurableQueue
2: Node* head;
3: Node* tail;
4: T* RVals[MAX];
5: DurableQueue():
6:   T* node := new Node(T());
7:   flush(node);
8:   head := node;
9:   flush(&head);
10:  tail := node;
11:  flush(&tail);
12:  forall i
13:    RVals[i] := null;
14:    flush(&RVals[i]);
```

Friedman et al. [FHMP18] adapt MSQ to a system comprising both VM and NVM. In such systems, computations take place in VM as normal, but data is periodically flushed to NVM by the system. Such flushes may happen at any time while the algorithm executes, and there is no specific (e.g., FIFO) order in which writes in VM are flushed to NVM. In addition to system controlled flushes, a programmer may introduce explicit **flush** events that transfer data from VM to NVM. Only data in NVM persists after a crash (e.g., power loss). A persistent data structure must enable recovery from such an event, as opposed to a full system restart. In doing this, it must ensure some notion of consistency in the presence of crashes and a subsequent recovery operation. Following Friedman et al. [FHMP18], the notion of consistency we use is durable linearizability (see Section 3).

The persistent queue uses the same underlying data structure as MSQ (see Algorithm 1), but nodes contain an additional field, `deqID` (initialised to -1), which stores the ID of the thread that removed the node from the queue. (The `deqID` field is further explained as part of the dequeue operation below.) In addition to the head and tail pointers, it uses an array of pointers, `RVals`, with one index for each thread, containing either `null` (which is the initial value), a pointer to a cell which itself either contains `empty` (which signifies that the thread last saw an empty queue), or a value (which is the value that was last dequeued). Unlike MSQ, the persistent dequeue operation does not return a value; instead the returned value for `tid` is stored in the cell pointed to by `RVals[tid]`.

Persistent enqueue. The basic structure (see Algorithm 2) is the same as the enqueue of MSQ. In addition, to ensure that the linked list data structure is recoverable after a crash, nodes and next pointers have to be persisted after being modified in VM.

This is achieved by using three **flush** operations in lines 3, 10 and 14. The first ensures that the node is persisted before it is inserted into the queue; the second and third ensure that the next pointer of a lagging tail pointer is persisted before the tail is advanced. Note that updates to tail do not need to be explicitly flushed because it can be recomputed during recovery by traversing the persistent list.

Persistent dequeue. The basic structure of the dequeue operation also resembles the dequeue of MSQ. In addition it uses variables `RVals` and `deqID` to guarantee durable linearizability. `RVals` is an array of pointers to cells that are used to store the value returned by each dequeue. A dequeue creates a new cell at Line 2, then flushes it at Line 3. The pointer to this cell is stored in `RVals` at Line 4, and this pointer is made persistent at Line 5.

The `deqID` field is used to logically mark nodes that are dequeued, which occurs at the successful CAS at Line 20. This logical dequeue is made persistent by flushing the `deqID` at Line 21. After a node has been logically dequeued, the dequeued value is stored in the cell pointed to by `RVals[tid]` (see Line 22) where `tid` is the thread ID of the dequeuing thread. This dequeued value is made persistent at Line 23. A dequeue by thread `tid` stores `empty` in `RVals[tid]` if the queue is empty in Line 13, and this value is made persistent at Line 14.

The persistent dequeue operation employs an additional helping mechanism to ensure that these new fields are made persistent in the correct order. In particular, a node that has been logically dequeued in VM must be made persistent before another dequeue is allowed to succeed.

Algorithm 2 Enqueue and dequeue methods of Friedman et al. [FHMP18]

```
1:  procedure Enq(T val)
2:  Node* node := new Node(val);
3:  flush(node);
4:  while true do
5:    Node* last := tail;
6:    Node* nxt := last→next;
7:    if (last = tail)
8:      if (nxt = null)
9:        if CAS(&last→next,nxt,node)
10:         flush(&last→next);
11:         CAS(&tail, last, node);
12:         return;
13:     else
14:       flush(&last→next);
15:       CAS(&tail, last, nxt);
```

```
1:  procedure Deq(int tid)
2:  T* newRVal := new T();
3:  flush(newRVal);
4:  RVals[tid] := newRVal;
5:  flush(&RVals[tid]);
6:  while true do
7:    Node* first := head;
8:    Node* last := tail;
9:    Node* nxt := first→next;
10:   if (first = head)
11:     if (first = last)
12:       if (nxt = null)
13:         *RVals[tid] := empty;
14:         flush(RVals[tid]);
15:         return;
16:       flush(&last→next);
17:       CAS(&tail, last, nxt);
18:     else
19:       T val := nxt→val;
20:       if CAS(&nxt→deqID,-1,tid)
21:         flush(&nxt→deqID);
22:         *RVals[tid] := val;
23:         flush(RVals[tid]);
24:         CAS(&head, first, nxt);
25:         return;
26:       else
27:         T* addr:=RVals[nxt→deqID];
28:         if (head = first)
29:           flush(&nxt→deqID);
30:           *addr := val;
31:           flush(addr);
32:           CAS(&head,first,nxt);
```

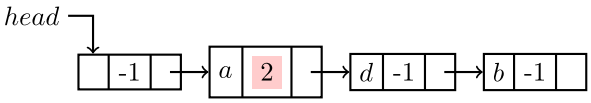


Fig. 1. State of the queue with contents $\langle a, d, b\rangle$; volatile data represented using shading; dequeue of thread 2 currently running

Therefore, if a thread recognises that `deqID` is not -1 at Line 20, it helps the other thread by flushing the `deqID` field, writing the dequeued value into the cell pointed to by `RVals[nxt→tid]`, flushing this cell, and finally advancing the head pointer. As an example consider the state of the queue depicted in Figure 1. The head pointer `head` (both in persistent and volatile memory) points to the sentinel node at the start. The queue currently contains elements a, d and b. Assume that a thread 2 has already started dequeing and has successfully executed Line 20 (thereby placing its id into the `deqID` field of the first node). This field has however not been made persistent yet. Now another thread (say, 5) starts dequeing. Its CAS at Line 20 fails (it sees the `deqID` entry to not be -1 anymore). Thus it starts helping: it flushes the 2 entry in the `deqID` field of the first node, updates the return value arrays entry of thread 2 and makes it persistent. Only then can thread 5 resume its own dequeuing operation with the next iteration of the while loop.

Note that the helping thread may be delayed between the read at Line 27 and the write at Line 30, and the original thread `tid` may begin a new dequeue operation in this interval. In this case, since `tid` allocates a fresh cell at Line 2, the helping thread's write at Line 30 will harmlessly modify a previous cell.

Recovery. After a crash, and prior to resuming normal operation, persistent data structures must perform a *recovery* operation that restores the state of the data structure in VM from NVM. The recovery procedure proposed by Friedman et al. is multithreaded (and complex), so we elide its details here (an interested reader may consult [FHMP18]). Instead, we provide a simpler single-threaded recovery operation (see Algorithm 4), which we describe in more detail in Section 6.1.

3. Durable linearizability

We now define *durable linearizability* [IMS16], a central correctness condition for persistent concurrent data structures. Like linearizability, durable linearizability is defined over *histories* recording the *invocation* and *response* events of operations executed on the concurrent data structure. Unlike linearizability, durably linearizable histories include crash events.

Formally, we let Σ be the set of operations. For a queue, $\Sigma = \{\texttt{Enq}, \texttt{Deq}\}$. A history is a sequence of events, each of which is either (a) an invocation of an operation op by a thread $t \in T$ with values $\vec{v}$, written $inv_t(op, \vec{v})$, (b) responses of op in thread t with value v, written $res_t(op, v)$, or (c) a system-wide crash c.

Given a history h, we let $ops(h)$ denote h restricted to non-crash events, and $h_{|t}$ denote h restricted to (non-crash) events of thread $t \in T$. The crash events partition a history into $h = h_0 c_1 h_1 c_2 ... h_{n-1} c_n h_n$, such that n is the number of crash events in h, c_i is the ith crash event and $ops(h_i) = h_i$ (i.e., h_i contains no crash events). We call the subhistory h_i the *i-th era* of h (i.e., all largest subsequences of h without crashes are eras). For a history h and events e_1, e_2, we write $e_1 <_h e_2$ whenever $h = h_0 e_1 h_1 e_2 h_2$.

A history h is said to be *sequential* iff every invocation event (except if it is the last event in h) is immediately followed by its corresponding response event; it is *well formed* if and only if (a) $h_{|t}$ is sequential for every thread t and (b) each thread id appears in at most one era. Any invocation that is not followed by its response event is called a *pending* invocation. We consider well-formed histories only. A history h defines a *happens-before* ordering on the events occurring in h by letting $e_1 \prec_h e_2$ iff $e_1 <_h e_2$ and e_1 is a response and e_2 an invocation event. Linearizability (and durable linearizability) requires a notion of a *legal history*, which we define using a sequential object.

Definition 3.1 (Sequential object) A *sequential object* over a base type Val is a 5-tuple $(\Sigma, S, s_0, in, \rho)$ where

- Σ is an *alphabet* of operations, S is a set of states and s_0 the initial state,
- $in : \Sigma \to \mathbb{N}$ is an *input function* telling us the number of inputs an operation $op \in \Sigma$ takes, and
- $\rho : S \times \Sigma \times Val^* \to S \times (Val \cup \{\texttt{empty}, \bot\})$ is a partial *transition function*.

We assume outputs of operations to consist of a single value which possibly is the symbol `empty` or no value denoted by $\bot$. In the following we let $\vec{v} = v_1 v_2 \dots v_n$ denote a string of n elements and write $\#\vec{v}$ to denote its length n. We write $inv_t(op, \vec{v})$ for an invocation of the operation op with $n = \#\vec{v}$ inputs by thread t and let Inv be the set of all such invocations. Similarly, we let Res be the set of all responses.

The *legal* histories of a sequential object $\mathbb{S} = (\Sigma, S, s_0, in, \rho)$ are defined as follows. We write

$$s \xrightarrow{inv_t(op,\vec{v})res_t(op,v)} s'$$

for $\rho(s, op, \vec{v}) = (s', v)$ and $t \in T$. For a sequence w of invocations and responses, we write $s \xrightarrow{w} s'$ iff either $w = \langle\rangle$ and $s = s'$, or $w = u \circ w'$ and there exists an s'' such that $s \xrightarrow{u} s''$ and $s'' \xrightarrow{w'} s'$. The set of *legal histories* of $\mathbb{S}$ is given by $legal_{\mathbb{S}} = \{w \in (\mathsf{Inv} \cup \mathsf{Res})^* \mid \exists\, \mathsf{s} \in \mathsf{S}.\ \mathsf{s}_0 \xrightarrow{w} \mathsf{s}\}$.

Example 3.1 A sequential queue, $\mathbb{Q}$, storing elements of type V is defined by $\Sigma = \{\texttt{Enq}, \texttt{Deq}\}$, $in(\texttt{Enq}) = 1$, $in(\texttt{Deq}) = 0$, $q_0 = \langle\rangle$, plus the transition function ρ given as a set of pairs out of $(S \times \Sigma \times Val^*) \times (S \times (Val \cup \{\texttt{empty}, \bot\}))$ is

$$\rho = \{\big((q, \texttt{Enq}, v), (q \cdot v, \bot)\big) \mid v \in V \wedge q \in V^*\} \cup \\ \{\big((v \cdot q, \texttt{Deq}, \varepsilon), (q, v)\big) \mid v \in V \wedge q \in V^*\} \cup \{\big((\langle\rangle, \texttt{Deq}, \varepsilon), (\langle\rangle, \texttt{empty})\big)\}$$

where ε is the empty string, $\langle\rangle$ is the empty sequence and $\cdot$ is used for sequence concatenation. For $\mathbb{Q}$, the history h below is sequential and legal

$$h = \langle inv_1(\texttt{Enq}, a), res_1(\texttt{Enq}, \bot), inv_2(\texttt{Deq}, \varepsilon), res_2(\texttt{Deq}, a)\rangle$$

whereas the history $h \cdot \langle inv_3(\texttt{Deq}, \varepsilon), res_3(\texttt{Deq}, b)\rangle$ is sequential but not legal.

For the definition of durable linearizability some more notation is needed. We write $h \equiv h'$ if $h_{|t} = h'_{|t}$ for all threads t. We let $compl(h)$ (the completion) be the set of histories that can be obtained from h by appending (some) missing responses at the end, and use $trunc(h)$ to remove pending invocations from a history h (or a set of histories).

Definition 3.2 (Linearizability [HW90]) A (crash-free) history h is *linearizable* if there is some $h' \in trunc(compl(h))$ and some legal sequential history h_S such that (i) $h' \equiv h_S$ and (ii) $\forall\, e_1, e_2 \in h' : e_1 \prec_{h'} e_2 \Rightarrow e_1 \prec_{h_S} e_2$.

For durable linearizability, this definition is now simply lifted to histories with crashes.

Definition 3.3 (Durable linearizability [IMS16]) A history h is *durably linearizable* if it is well formed and $ops(h)$ is linearizable.

Informally, durable linearizability guarantees that even after a crash the state of the concurrent object remains consistent with the abstract specification. This means that the effect of any operations that completed before a crash are preserved after the crash. The effect of operations that did not complete before a crash may or may not be preserved. For example, the concurrent history

$$hc = \langle inv_1(\texttt{Enq}, a), inv_3(\texttt{Deq}, \varepsilon), res_1(\texttt{Enq}, \bot), c, inv_2(\texttt{Deq}, \varepsilon), res_2(\texttt{Deq}, a)\rangle$$

is durably linearizable since $ops(hc) = \langle inv_1(\texttt{Enq}, a), inv_3(\texttt{Deq}, \varepsilon), res_1(\texttt{Enq}, \bot), inv_2(\texttt{Deq}, \varepsilon), res_2(\texttt{Deq}, a)\rangle$ is linearizable with respect to the history h in Example 3.1. On the other hand the history

$$\langle inv_1(\texttt{Enq}, a), inv_3(\texttt{Enq}, b), res_1(\texttt{Enq}, \bot), c, inv_2(\texttt{Deq}, \varepsilon), res_2(\texttt{Deq}, \texttt{empty})\rangle$$

is not durably linearizable since the effect of the completed operation $\texttt{Enq}(a)$ is not preserved after the crash.

Our methodology for proving durable linearizability does not use Definition 3.3 directly; instead it uses the following characterisation, which defines the set of all durably linearizable histories for a sequential object.

We let $\textsc{Lin}(\mathbb{S})$ be the set of histories linearizable wrt. the legal histories of sequential object $\mathbb{S}$ and define

$$\textsc{DurLin}(\mathbb{S}) = \{h \in (\mathsf{Inv} \cup \mathsf{Res} \cup \{\mathsf{c}\})^* \mid ops(\mathsf{h}) \in \textsc{Lin}(\mathbb{S})\}$$

For a given concurrent durable data structure implementing a sequential object $\mathbb{S}$, proving its correctness thus amounts to showing that all histories of the implementation are in $\textsc{DurLin}(\mathbb{S})$. To this end, for a given $\mathbb{S}$, we develop an operational model $\textsc{DurAut}(\mathbb{S})$ whose behaviours generate $\textsc{DurLin}(\mathbb{S})$. We then use a standard refinement approach to show that the implementation model is a refinement of $\textsc{DurAut}(\mathbb{S})$. This is enough to guarantee that the original implementation is durably linearizable.

4. An operational model for durable linearizability

The operational model for durable linearizability is formalised in terms of an *Input/Output automaton (IOA)* [LT87]. This framework is often used for proving linearizability via refinement [DD15], and here we intend to similarly employ it for proving durable linearizability.

Definition 4.1 An *Input/Output automaton* (IOA) is a labeled transition system A with

- a set of *states* $states(A)$,
- a set of *start* states $start(A) \subseteq states(A)$,
- a set of *actions* $acts(A)$, and
- a *step* (or transition) relation $step(A) \subseteq states(A) \times acts(A) \times states(A)$ (so that the actions label the steps).

The set $acts(A)$ is partitioned into *internal* actions, $internal(A)$ and *external* actions, $external(A)$.[1] The internal actions represent events of the system that are not visible to the environment, whereas the external actions represent the automaton's interactions with its environment.

[1] In the standard IOA setting, external actions are further subdivided into input and output actions; this distinction is not needed for this current work.

An *execution* of an IOA A is a sequence $\sigma = s_0 a_1 s_1 a_2 s_2 a_3 \ldots$ of alternating states and actions such that $s_0 \in start(A)$ and for each i, $(s_i, a_{i+1}, s_{i+1}) \in step(A)$. A *reachable* state of A is a state appearing in an execution of A. We let $reach(A)$ denote the set of reachable states of A. An *invariant* of A is any superset of the reachable states of A (equivalently, any predicate satisfied by all reachable states of A). A *trace* of A is any sequence of (external) actions obtained by projecting onto the external actions of any execution of A. The set of traces of A, $traces(A)$, represents A's externally visible behaviour. If every trace of an automaton C is also a trace of an automaton A, then we say that C *implements* or *refines* A.

One advantage of using IOA (and related operational models) is that we can prove refinement using the method of *simulations*. To prove refinement in our case study, we will later employ *forward simulation*. The definition of forward simulation we use is adapted from that of Lynch and Vaandrager [LV95].

Definition 4.2 A *forward simulation* from a concrete IOA C to an abstract IOA A is a relation $abs \subseteq states(C) \times states(A)$ such that each of the following holds.

Initialisation $\forall\, cs \in start(C).\ \exists\, as \in start(A).\ abs(cs, as)$

External step correspondence

$\forall\, cs \in reach(C),\ as \in reach(A),\ a \in external(C),\ cs' \in states(C).$
$abs(cs, as) \wedge (cs, a, cs') \in step(C) \Rightarrow \exists\, as' \in states(A).\ abs(cs', as') \wedge (as, a, as') \in step(A).$

Internal step correspondence

$\forall\, cs \in reach(C),\ as \in reach(A),\ a \in internal(C),\ cs' \in states(C).$
$abs(cs, as) \wedge (cs, a, cs') \in step(C) \Rightarrow$
$abs(cs', as) \vee \exists\, a' \in internal(A),\ as' \in states(A).\ abs(cs', as') \wedge (as, a', as') \in step(A).$

Forward simulation is *sound* in the sense that if there is a forward simulation between A and C, then C refines A.

It is well known that forward simulation is not complete for proving refinement. We also require a notion of simulation known as *backward simulation* [LV95]. In this paper, we use backward simulation to prove refinement between the abstract specification (see Fig. 2) and an intermediate automaton (see Fig. 3). Such techniques are well documented in the literature (see [DGLM04, DD15]) and their details are largely uninteresting for the purposes of our case study. We therefore elide the definition of backward simulation in this paper.

For an arbitrary sequential object $\mathbb{S}$, we next construct a durable automaton DurAut($\mathbb{S}$) (see Fig. 2) whose traces are histories in DurLin($\mathbb{S}$) only. This automaton can serve as a specification automaton in a refinement proof. The state of this automaton incorporates the state s of the sequential object $\mathbb{S}$, plus for every thread $t \in T$:

- a program counter fixing whether the thread is still *idle*, is *ready* to be started, is *crashed* (i.e., has been active during a crash), or is currently executing an operation,
- possible input values of the thread's operations and a possible output value.

The step relation of the automaton is – as usual – given in the form of pre- and postconditions of actions. For every operation *op* in the sequential object, the automaton has actions $inv(op)$, $do(op)$ and $res(op)$, where $do(op)$ corresponds to execution of the abstract operation *op*, potentially changing the state of the sequential object. We use $inv_t(op, \vec{v})$ and $res_t(op, v)$ for $inv(op)_t(\vec{v})$ and $res(op)_t(v)$, respectively.

Note that a thread may only invoke an operation if it is $ready$. We furthermore have a dedicated $crash$ action that may be executed at any time that sets all active threads to $crashed$. To ensure that crashed threads are confined to a single era, we use a separate action run that enables idle threads to become $ready$. While $inv(op)$, $res(op)$ and $crash$ are external actions, run and $do(op)$ are internal.

The theorem below ensures that the traces of the durable automaton are exactly the durably linearizable histories of $\mathbb{S}$.

Theorem 4.1 Let $\mathbb{S}$ be a sequential object. Then $traces(\text{DurAut}(\mathbb{S})) = \text{DurLin}(\mathbb{S})$.

Proof. We start by showing that $traces(\text{DurAut}(\mathbb{S})) \subseteq \text{DurLin}(\mathbb{S})$. Let $\sigma = cs_0 a_1 cs_1 \ldots a_n cs_n$ be an execution of DurAut($\mathbb{S}$) and let $cs_i.s$, $cs_i.out$ etc. be the components of state cs_i. Let tr be the trace of σ. We construct the history h by making the following changes to tr (in this order).

Completion For every a_i being a do action $do_t(op)$ in σ without matching $res_t(op)$, we add $res_t(op, v)$ such that $v = cs_i.out(t)$ to the end of tr.

$inv_t(op, \vec{v})$
Pre: $pc(t) = ready$
$\#\vec{v} = in(op)$
Eff: $pc(t) := inv(op)$
$\vec{val}(t) := \vec{v}$

$do_t(op)$
Pre: $pc(t) = inv(op)$
Eff: $pc(t) := res(op)$
$(s, out(t)) := \rho(s, op, \vec{val}(t))$

$res_t(op, v)$
Pre: $pc(t) = res(op)$
$v = out(t)$
Eff: $pc(t) := ready$

run_t
Pre: $pc(t) = idle$
Eff: $pc(t) := ready$

$crash$
Pre: $true$
Eff: $pc := \lambda t : T.\mathbf{if}\ pc(t) \neq idle\ \mathbf{then}\ crashed\ \mathbf{else}\ pc(t)$

$$states(A) : pc : T \to \{idle, ready, crashed\} \cup \{inv(op), res(op) \mid op \in \Sigma\}$$
$$s : S \text{ (the state of the sequential object)}$$
$$\vec{val} : T \to Val^* \text{ (values of inputs)}$$
$$out : T \to Val \text{ (the value of the output)}$$
$$start(A) : \forall t \in T : pc(t) = idle \wedge \vec{val}(t) = \epsilon \wedge out(t) = \epsilon \wedge s = s_0$$

Fig. 2. Durable automaton $A = \text{DURAUT}(\mathbb{S})$ for a sequential object $\mathbb{S} = (\Sigma, S, s_0, in, \rho)$

Truncation We remove all $inv_t(op, \vec{v})$ without matching response.

Next, we need to construct a legal sequential history h_S such that $ops(h) \equiv h_S$. Let $i_1, \ldots, i_k$ be the indices of σ such that a_{i_j} is a do action $do_t(op)$. Then $\rho(cs_{i_{j-1}}.s, op, \vec{v}) = (cs_{i_j}.s, cs_{i_j}.out(t))$ by definition of the durable automaton. We set

$$w_{i_j} = inv_t(op, \vec{v})\, res_t(op, cs_{i_j}.out(t)).$$

We let $h_S = w_{i_1} \ldots w_{i_k}$ and $h_S \in legal(\mathbb{S})$.

Now assume $e_1 \prec_h e_2$. By definition, $e_1 = res_t(op, v)$ and $e_2 = inv_{t'}(op', \vec{v})$ for some $t, t' \in T$. Then e_1 has not been added to the trace tr by completion since responses are added to the end. By construction of the durable automaton threads execute inv, do and res operations in this ordering only. Hence the execution σ contains an action $do_t(op)$ prior to e_1 and an action $do_{t'}(op')$ following e_2. Hence $e_1 \prec_{h_S} e_2$.

We next proceed with showing that $traces(\text{DURAUT}(\mathbb{S})) \supseteq \text{DURLIN}(\mathbb{S})$. Let h be a history of $\text{DURLIN}(\mathbb{S})$, i.e., $ops(h) \in \text{LIN}(\mathbb{S})$. By the definition of linearizability there is hence a history $h' \in trunc(compl(ops(h))$ and some h^s which is a legal, sequential history of $\mathbb{S}$ such that for all $e_1, e_2 \in h'$: $e_1 \prec_{h'} e_2$ implies $e_1 \prec_{h^s} e_2$.

Let $h^s = w_0 \ldots w_m$ and assume $s_0 \xrightarrow{w_0} s_1 \ldots s_{m-1} \xrightarrow{w_m} s_m$ are the transitions in the sequential object $\mathbb{S}$.

We inductively construct an execution $\sigma = st_0 a_1 st_1 a_2 \ldots$ of the durable automaton such that $traces(\sigma) = h$. This construction maintains a prefix σ_i of σ, remainder histories h_i (suffix of h) and h_i^s (suffix of h^s) and a set $crashed_i \subseteq T$. We furthermore maintain the following relations among these variables:

$$\forall\, t \in crashed_i.\ st_i.pc(t) = crashed \tag{1}$$
$$st_i.s = s_i \tag{2}$$

and t is in $crashed_i$ if there exists a prefix $\hat{h}$ such that $h = \hat{h} \cdot \langle crash \rangle \cdot h_i$ and t has invoked an operation in $\hat{h}$.

Initially, $\sigma_0 := st_0$ s.t. $st_0 \in start(A)$ (which is unique), $crashed_0 := \emptyset$, $h_0 := h$, $h_0^s := h^s$.

The so far constructed execution $\sigma_i = st_0 a_1 \ldots a_i st_i$ is now extended in the following way according to the next event in the history h_i.

- Next event is an invocation: $h_i = \langle inv_t(op, v) \rangle \cdot h_i'$.
  Since each thread appears at most once in one era, $t \notin crashed_i$. By well-formedness of histories, $st_i.pc(t) \in \{idle, ready\}$.

  If $st_i.pc(t) = idle$: extend σ_i by $run_t\, st_{i+1}$ (st_{i+1} according to transition relation of automaton) and set h_{i+1} to h_i, $crashed_{i+1}$ to $crashed_i$, h_{i+1}^s to h_i^s.

  If $st_i.pc(t) = ready$: extend σ_i by $inv_t(op, v) st_{i+1}$ (st_{i+1} according to transition relation of automaton), set h_{i+1} to h_i', $crashed_{i+1}$ to $crashed_i$, h_{i+1}^s to h_i^s.

- Next event is a crash: $h_i = \langle c \rangle \cdot h'_i$.
  Then extend σ_i by $crash\ st_{i+1}$ (st_{i+1} according to transition relation of automaton), set h_{i+1} to h'_i, $crashed_{i+1}$ to $crashed_i \cup \{t \mid st_i.pc(t) \neq idle\}$, h^s_{i+1} to h^s_i.
- Next event is a response: $h_i = \langle res_t(op, v) \rangle \cdot h'_i$.
  Here, we again need to consider two cases:
  1. $h^s_i = inv_{t'}(op', u) res_{t'}(op', u') \cdot h''$, $t \neq t'$, i.e. t' linearizes op' before t linearizes op.
     Then we extend σ_i by $do_{t'}(op') st_{i+1}$ (st_{i+1} according to transition relation of automaton), set h_{i+1} to h_i, $crashed_{i+1}$ to $crashed_i$, h^s_{i+1} to h''.
     We can do so because $st_i.pc(t') = inv(op')$ (by well-formedness of history and linearizability preserving real-time order) and $st_i.val(t') = u$ and $\rho(st_i.s, op', \vec{val}(t')) = (st_{i+1}.s, u')$ (by linearizability preserving thread operations and the definition of sequential object).
  2. $h^s_i = inv_t(op, u) res_t(op, v) \cdot h''$
     Then we extend σ_i by two actions: $do_t(op) st_{i+1} res_t(op, v) st_{i+2}$ (st_{i+1}, st_{i+2} according to transition relation of automaton), set h_{i+1} and h_{i+2} to h'_i, $crashed_{i+1}$ and $crashed_{i+2}$ to $crashed_i$, h^s_{i+1} to h''.

By construction, $traces(\sigma) = h$. □

5. Proof approach

We verify durable linearizability by proving refinement between the implementation model and DurAut($\mathbb{Q}$) (the queue $\mathbb{Q}$ as of Example 3.1) using the IOA formalism introduced in Section 4. To this end, a number of further steps are required which we outline here before we go into more detail in later sections.

Selection of simulation type (this section). Refinement on IOA can be shown by *forward* or *backward simulations* [LV95] which are both sound (and in combination complete) for showing refinement. For our case study, we need to determine the appropriate simulation type.

Translation of implementation (Section 6). Concurrent algorithms are typically directly given in a programming language, which for the durable queue is C. Hence, we need to translate C programs to IOA. In this, we in particular need to take care of the fact that we have volatile as well as persistent memory.

Proofs of invariants (Section 7). Simulation proofs often first of all require the definition and proof of invariants on the implementation. These invariants describe constraints on the reachable states of the concurrent data structure. We aim at *thread-local* proofs of invariants.

We will exemplarily describe all the steps along our case study of the persistent queue. Before going into more detail about these steps, we first take a look at the simulation question. Forward and backward simulations allow a step-by-step comparison between the operations of the implementation and the abstract specification using an abstraction relation. Forward simulations are often easier to show than backward simulations. However, like for the MSQ [DGLM04, DD15], we require a backward simulation for the persistent queue here. To split the complexity of proving, we in addition introduce an *intermediate automaton*. For the proof, we establish a backward simulation between the intermediate automaton and DurAut($\mathbb{Q}$) as well as a forward simulation between the implementation of the persistent queue and the intermediate automaton. The intermediate automaton resolves non-determinism at the abstract level in the same way as used in existing proofs of MSQ. Since refinement is transitive and guarantees trace inclusion, these two simulation proofs together are sufficient to show that the persistent queue is durably linearizable.

In Fig. 3 we present the intermediate automaton *IDQ* which is used as an intermediate level in the overall refinement proof. The automaton *IDQ* is similar to the durable automaton for the queue datatype, DurAut($\mathbb{Q}$) (see Fig. 2 instantiated for the queue from Example 3.1). Note that we have specified one generic invocation operation $inv_t(op, \vec{v})$ for both *Enq* and *Deq*. As with DurAut($\mathbb{Q}$), it has variables pc, val and out, which play the same role, and variable q instantiates the state s. Furthermore, all its actions except for $checkEmp$ are also actions of DurAut($\mathbb{Q}$), and have essentially the same effect. For *IDQ* we have the following property[2].

[2] For the proof in KIV, see [KIV20].

$inv_t(op, \vec{v})$
Pre: $pc(t) = ready$
$\#\vec{v} = in(op)$
Eff: $pc(t) := inv(op)$
$\vec{val}(t) := \vec{v}$
$obsEmp(t) := false$

$do_t(op)$
Pre: $pc(t) = inv(op)$
Eff: $pc(t) := res(op)$
$(q, out(t)) := \rho(q, op, \vec{val}(t))$

$res_t(Enq)$
Pre: $pc(t) = res(Enq)$
Eff: $pc(t) := ready$

$res_t(Deq, v)$
Pre: $pc(t) = res(Deq)$
$v = out(t)$
Eff: $pc(t) := ready$

$do_t(Deq)$
Pre: $pc(t) = inv(Deq)$
$obsEmp(t)$
Eff: $pc(t) := res(Deq)$
$out(t) := \texttt{empty}$

$checkEmp_t$
Pre: $pc(t) = inv(Deq)$
$q = \langle\rangle$
Eff: $obsEmp(t) := true$

run_t
Pre: $pc(t) = idle$
Eff: $pc(t) := ready$

$crash$
Pre: $true$
Eff: $pc := \lambda t : T.\mathbf{if}\ pc(t) \neq idle\ \mathbf{then}\ crashed\ \mathbf{else}\ pc(t)$

$states(IDQ) : pc : T \to \{idle, ready, crashed\} \cup \{inv(op), res(op) \mid op \in \Sigma\}$
$q : V^*$ (the state of the sequential queue)
$val : T \to V^*$ (values of inputs)
$out : T \to V \cup \{\bot\}$ (the value of the output)
$obsEmp : T \to \mathbb{B}$

Fig. 3. The intermediate automaton *IDQ*

Theorem 5.1 $traces(IDQ) \subseteq traces(\text{DURAUT}(\mathbb{Q}))$.

The additional features of *IDQ* exist to model a behaviour where a dequeue thread first *observes* that the queue is empty, and later decides to return `empty`, at a point when the queue may no longer *be* empty. The observation is modelled by a $checkEmp_t$ action, which records in the $obsEmp(t)$ variable the fact that the queue was empty during the execution of t's dequeue operation. In this automaton, it is possible for a thread t to execute a $do_t(Deq)$ transition and set the output value to `empty` whenever $obsEmp(t)$ has been set to $true$. We note that the queue may not actually be empty when this transition takes place, but this does not affect soundness of the proof method since $obsEmp(t)$ being set to $true$ indicates that the queue has been empty at *some point* during the operation's execution. Further details of this technique, in the context of linearizability, may be found in [DGLM04, DD15, DSW11].

This theorem already establishes the first half of our refinement proof, showing that the intermediate automaton refines DURAUT($\mathbb{Q}$). We next look at the other half, proving the implementation of the persistent queue to refine the intermediate automaton.

6. Translating C programs to input/output automata

This section describes the translation of the implementation to IO automata. Once we have an IOA, we can establish a refinement relationship between the implementation IOA and the intermediate automaton using proof techniques for IOA refinement. The implementation IOA describes the states, actions and in particular the step relation $step \subseteq State \times Action \times State$. The notation used for specifying IOAs is the notation used in the theorem prover KIV, i.e., the step relation is given via predicate logic axioms.

In earlier case studies, we have done such translations manually [SDW14, DSW11, DDD+16]. For programs with only a few steps this is not too much work, but for longer programs the translation of atomic program steps (the programs here have ca. 50 atomic steps) to axioms of predicate logic becomes error prone. Therefore the translation is now done in two steps. First the C programs are translated manually to concurrent programs as used by KIV's program logic. The control structures of the C programs can be used almost one-to-one, however, care must be taken about the granularity of atomic steps and how data structures are specified in the theorem prover. In particular, the volatile and persistent heap that is implicit in the C programs must be explicitly specified. We describe this translation in Section 6.1. The result for the dequeue operation is shown

in Algorithm 3. The algorithm includes annotations used by the refinement proof, shown in gray, which will be explained later.

The second step is a comprehensive framework which automatically translates such programs as well as specifications of atomic steps like flushes and crash, or the ones given for the intermediate automaton of Fig. 3, to predicate logic axioms which define the transition relation of an IOA. Section 6.2 defines the resulting structure of the $State$ and $Action$ type, and Section 6.3 breaks down the definition of the full transition relation to predicate logic axioms.

Together with an automated generation of thread-local proof obligations from assertions for program points given in Section 7, the framework allows one to focus verification of the refinement.

6.1. The KIV model for C programs

The KIV model for the C algorithm shown in Algorithm 3 has to axiomatise data types for all types used in C. The main effort is needed to translate the implicit heap structure used in C into an explicit data structure.

One natural way to represent a heap is as a (partial) map from locations to values. The main difficulty in using this technique is that such a partial map has a single fixed range type, whereas the heap we are modeling stores various different types, such as references and boolean flags. This problem is well-known, and various solutions have been proposed (e.g. see [TK05]). We use a simple technique based on representing heaps as partial maps, but we use several such maps in a way that enables us to exploit the KIV type system as much as possible.

Our KIV model uses a type Ref to represent references to heap allocated objects and a type $Node$ to represent the nodes themselves. The model has a state variable $vs : Ref \rightharpoonup Node$ containing the *volatile* memory version of the queue nodes, and a corresponding map $ps : Ref \rightharpoonup Node$ containing the *persistent* memory version. Retrieving the value stored under a reference r allocated in heap vs, is denoted as $vs[r]$. Every shared variable in our model except `Tail` follows this pattern: we have one variable storing the value currently in volatile memory, and one variable storing the value currently in persistent memory. The remaining variables are as follows:

- A variable $vhead{:}Ref$ containing the volatile value of *head*, and its persistent counterpart $phead : Ref$. In contrast to C, the KIV model does not store these values under a fixed address in the heap.
- A function $vRetRefs{:}Tid \rightarrow Ref$ and its persistent counterpart $pRetRefs : Tid \rightarrow Ref$. These variables represent the return-value array, which maps each thread to a cell on the heap for storing the thread's return value.
- A map $vrvHp{:}Ref \rightharpoonup OptValue$ and its persistent counterpart *prvHp*. These variables represent the return-value heap cells themselves. Each value in the type $OptValue$ is either `empty`, (when the cell has not yet been initialised) or $\texttt{Val}(v)$ for some value v (when the cell has been set to contain v).
- Our model has a variable $vtail : Ref$ representing the volatile value of `Tail`, with no persistent counterpart. This exception is because the queue algorithm never explicitly flushes `Tail`, and does not depend on the value of `Tail` being preserved across crashes. As we shall see, $vtail$ is set to an arbitrary value during crash events and then an appropriate value is inferred for $vtail$ during recovery.

Thus, the parts of the heap that contain queue nodes, return values, global variables, and the array containing references to return values are all stored as different variables. This strategy allows our accesses to the heap to be properly typed by simply typed lambda calculus, which is KIV's type system.

Recall that both threads and the NVM system are able to flush values from the volatile heap to persistent memory. Conceptually, these flushes are uniform: the value of any given location is flushed, irrespective of its type. However, because we use distinct variables to represent entities stored on the heap, we must define a flush action for each such pair. Thus,

- We model flushes of `Head` by performing the assignment $phead := vhead$.
- We model flushes of entries in the return-value array with the assignment $pRetRefs(t) := vRetRefs(t)$, where t is the ID of the thread whose entry is flushed.
- We model flushes of a return value stored at address r by performing the assignment $prvHp[r] := vrvHp[r]$ where r is the reference to the return value that is flushed.

Algorithm 3 Dequeue Operation in KIV

```
ready : DEQ(;optval) {
D1:   optval := empty;
D2:   choose* ref, oval with ref ∉ vrvHp ∧ ref ∉ prvHp ∧ ref ≠ null in {
        vrvHp[ref] := oval; prvHp[ref] := oval;
        rvowns(ref) := Pending(t);
D3:     vretRefs(t) := ref;
D4:     pretRefs(t) := vretRefs(t) ; /* flush */
D5:     let success = false in
D6:      while ¬ success do {
D7:       let first = vhead in
D8:        let last = vtail in
D9: with (qrl = [] ⊃ checkEmp_t; τ):
          let nxt = (ps ⊕ vs)[first].next in
D10:       if first = vhead then { /* else D33 */
D11:        if first = last then {/* else D18 */
D12:          if nxt = null then { /* else D16 */
D13:            vrvHp[ref] := empty;
D14: with do_t(Deq):
                prvHp[ref] := vrvHp[ref]; /* flush */
D15:            success := true, optval := empty; /* finish loop */
              } else { /* nxt ≠ null*/
D16: with (last = (orl + qrl).last ⊃ do_owns(nxt)(Enq); τ):
                ps[last].next := vs[last].next, /* flush */
                qrl := (last = (orl + qrl).last ⊃ qrl + nxt; qrl),
                owns := (last = (orl + qrl).last ⊃ owns(nxt := -1); owns);
D17:            if* vtail = last then vtail := nxt else skip/* CAS */
                /* restart loop */
            }} else { /* first ≠ last */
D18:          optval := Val((ps ⊕ vs)[nxt].val);
D19:          if* (ps ⊕ vs)[nxt].deqID = -1 /* CAS */
              then success := true,
                   vs[nxt].deqID := t,
                   rvowns(vretRefs(t)) := Claimed(t, optval.val);
              else success := false;
D20:          if success then { /* else D25 */
D21: with (ps[nxt].deqID = -1 ⊃ do_t(Deq);τ):
                ps[nxt].deqID := vs[nxt].deqID, /* flush */
                orl := (ps[nxt].deqID = -1 ⊃ orl + qrl.head; orl),
                qrl := (ps[nxt].deqID = -1 ⊃ qrl.tail; qrl);
D22:            vrvHp[ref] := optval;
D23:            prvHp[ref] := vrvHp[ref]; /* flush */
D24:            if* vhead = first then vhead := nxt else skip/* CAS */
                /* exit loop */
              } else { /* sucess = false */
D25:            let othert = (ps ⊕ vs)[nxt].deqID in
D26:             let addr = vretRefs(othert) in
D27:              if (vhead = first) then { /* else D32 */
D28: with (ps[nxt].deqID = -1 ⊃ do_othert(Deq); τ):
                  ps[nxt].deqID := Some(othert), /* flush */
                  orl := (ps[nxt].deqID = -1 ⊃ orl + qrl.head; orl),
                  qrl := (ps[nxt].deqID = -1 ⊃ qrl.tail; qrl);
D29:              vrvHp[addr] := optval;
D30:              prvHp[addr] := vrvHp[addr]; /* flush */
D31:              if* vhead = first then vhead := nxt else skip /* CAS */
                  /* restart loop */
                }
D32:              else skip /* vhead ≠ first */
           }}
D33:      else skip} /* first ≠ vhead */
D34: return ready;
```

Algorithm 4 Crash and Recovery

```
C1: procedure CRASH
C2: vRetRefs := ? , vrvHp := ?, vhead := ?, tail := ?;          ▷ randomise volatile state
C3: pcf := (λ t. if pcf(t) = idle then idle else crashed);
C4: lsf :∈ (λ t. lsf(t).tid = t);                    ▷ choose lsf st. tid = t for every thread t
C5: vs := ps, vRetRefs := pRetRefs, vhead := phead;   ▷ initialise volatile with persistent state
C6: owns := (λ r. None);                          ▷ ownership in auxiliary state must be reset
                                  ▷ recovery program (augmented with auxiliary code for orl, qrl)
C7: while ps[vhead].nxt ≠ null ∧ ps[ps[vhead].nxt].opttid ≠ None do
C8:   orl := orl + vhead
C9:   vhead := ps[vhead].nxt
C10: tail := vhead;
C11: while ps[tail].nxt ≠ null do
C12:   qrl := qrl + tail
C13:   tail := ps[tail].nxt
```

The code has explicit flushes as part of the enqueue and dequeue programs, as indicated by the comments for Algorithm 3. All flushes are additionally possible as steps of the operating system which can happen independently of any code the individual threads currently execute. These are modeled additionally as *global* steps in the KIV specification, which are added to the step relation of the IOA.

Modelling of crash and recovery. Recall that after a crash and restart, we assume NVM systems copy the persistent store back into volatile memory and run a queue-specific recovery procedure before starting any program threads. We model this three-phase *crash-copy-recover* process as follows (Algorithm 4 on page 13 presents the formal KIV description). In the first phase, we model the effect of the crash itself. First, all the shared variables are set to arbitrary values. Second, all threads that are not idle are moved to a crashed state, precisely as in the canonical automaton. Third, the local variables of each thread are set to nondeterministically chosen values[3] (using the notation $:\in$).

In the next phase our model copies the persistent variable of each persistent/volatile pair into its volatile counterpart. For example, the crash action performs the assignments $vhead := phead$ and $vs := ps$.

In the final (recovery) phase, our model executes a recovery procedure to bring the queue into a consistent state (see Algorithm 4). We assume that the NVM system runs the recovery procedure before starting any program threads. Therefore, no program thread using the queue executes any action between the crash event and the completion of the recovery procedure. For this reason, we execute the entire crash-copy-recovery process as a single *atomic* step, i.e., all statements of Algorithm 4 conceptually occur in one step (no concurrent activity possible).

Allocation of references. Our model must support the allocation of new references. To perform an allocation, a thread nondeterministically chooses a reference not already in the domain of the appropriate partial map, and then updates the map so that the new reference is mapped to some initial value specified in the code. Note that the queue algorithm does not explicitly deallocate any reference, depending on garbage collection or a similar mechanism. Accordingly, we do not model explicit deallocations. Thus, the domain of each partial map increases monotonically. To ensure that an allocation is always possible we require that the Ref type is infinite, while the domain of each map is finite in any reachable state.

KIV model. The full specification which contains all the programs and the axioms that are generated from them can be found online at [KIV20]. Here, we only give some more information on the notation used in Algorithm 3. The first items address the data types used, the rest explains the control structures.

- KIV prefers a direct specification of finite maps as a (non-free) datatype which is constructed from the empty map $\emptyset$ by adding or overriding key-value pairs with an update operator $vs[r := nd]$. The constructor yields a map, where reference r has been updated or allocated with value nd.
- The check that a reference r is allocated in vs, i.e. in the domain of the partial function vs, is written as $r \in vs$. Reading the node stored at reference $r \in vs$ is written as $vs[r]$.

[3] There is one exception here. Each thread t can access its own thread id via a local tid variable, which is preserved to be t.

- Reading a reference r from memory is always done from the combined memory ($ps \oplus vs$).[4] The result of $(ps \oplus vs)[r]$ is $vs[r]$ when $r \in vs$, otherwise $ps[r]$.
- Nodes nd of type $Node$ are specified as a typed tuple with three fields of type $Value$, $Tid \cup \{-1\}$, and Ref. The fields are selected (and overwritten by assignments) using selectors $nd.\texttt{val}$, $nd.\texttt{next}$ and $nd.\texttt{deqID}$. Flushing the next field of the node stored at r is written as the assignment $ps[r].\texttt{next} := vs[r].\texttt{next}$ and similar for the other two fields.
- The dequeue program starts with program counter value ready which indicates that the thread is not running a program. It also returns to this label at the end of the code. The program has no input parameters (before the semicolon), and one output parameter $optval$, which can either be $\texttt{Some}(v)$, where v is of type $Value$, or $\texttt{empty}$.
- Assignments separated by comma are executed as one atomic, parallel assignment.
- Allocation in line D2 is done by choosing a reference ref (the star indicates, that the choice is executed atomically with the next assignments) that is not yet allocated and by storing some random value $oval$ under this reference. Allocation must be done atomically in both the persistent and the volatile memory, to avoid inconsistencies.
- The "**if** CAS" of the C algorithm has been broken up into two atomic steps at lines D19 and D20 using a local variable $success$. The first step executes the CAS itself expanded to a conditional. Again the **if*** indicates that the (compare) test is executed together with the assignment in both branches as one atomic step. The second step executes the conditional around the CAS using the result $success$. The local variable is additionally used as the test when to leave the main while loop. This avoids having a return in the middle of the code. Several more statements have also been split to ensure atomicity, e.g. line D25 reads the threadid $othert$ into a local variable before writing it.
- A few **else skip** statements have been added, since the language currently does not support conditionals without an else part.

6.2. States and actions of the automaton

The main work of automatically translating the programs of the case study to an IOA is to generate axioms for the step relation $step \subseteq State \times Action \times State$ of the IOA from the individual atomic steps of the given enqueue and dequeue algorithms in Fig. 2. The KIV specification in Algorithm 3 (in contrast to Alg. 4) defines a *non-atomic* program[5]. In non-atomic KIV programs every statement bringing the program counter from one label to the next forms one *atomic* step. The labels are defined as an enumeration type PC of program counter values. The enumeration includes constants for threads being at some atomic step within a program but also idle for a thread that has not started, ready for a thread that has started but is in between calls of enqueue and dequeue, and crashed.

In the IO automaton such atomic steps are (most of the time) translated to internal τ actions. However, some such steps correspond to *persistence points*, i.e., the points in time when the effect of an operation (like the dequeue) becomes visible to other threads. The identification of such points are crucial for the later refinement proof (see Section 8) and the IOA actions of these steps have to be non-τ actions. KIV allows to specify the IOA action associated with a statement in the KIV model using *with-clauses*. An example of this can be seen in line D14 of Algorithm 3 which specifies the statement to be the $do_t(Deq)$ action. A more complex example of a with-clause is discussed later.

Besides such atomic steps of nonatomic programs, the translation to an IOA also has to consider two other types of steps: (1) Atomic steps of one thread, like starting a thread (moving from label idle to ready) as well as all the steps of IDQ shown in Fig. 3 that are labeled with a thread t, and (2) atomic *global* steps not executed by a specific thread, like system flushes and the crash-recovery. The next section explains the translation of all such steps to a step relation; this section fixes how the types $State$ and $Action$ are defined.

The states of the automaton are constructed as tuples $\mathsf{mkstate}(gs, lsf, pcf)$ with three components.

[4] Note that the occurrences of $ps \oplus vs$ could be simplified to just vs since we do not model cache evictions for $vs[r]$ when $vs[r] = ps[r]$. However, we leave open the possibility of cache evictions in future work, in which case $ps \oplus vs$ must be used.

[5] This can be specified in KIV.

- The global state gs (of type GS) is a tuple of all the *global* variables used. In the case study this state contains e.g. the *vhead* variable and the persistent and volatile heap ps and vs. We use $gs.\texttt{vhead}$, $gs.\texttt{ps}$ as selectors ("getters" in Java) for components. The global state also contains the auxiliary variables used for verification.
- The second component ls collects all the *local* variables used by the threads. Formally it is a function function $lsf : Tid \to LS$, where a tuple $ls : LS$ collects all the local variables used in programs. These include parameters (such as the input value to enqueue) as well as all local variables like $first$ and nxt used by the algorithms.
- The thread id t of the thread itself is always stored in local variable tid of each local state. This component can be read, but not assigned, resulting in the invariant $lsf(t).\texttt{tid} = t$. We use the notation $lsf(t := ls)$ to denote the function lsf, where $lsf(t)$ has been modified to be ls.
- The third component of the state pcf specifies the program counter for each thread. This component could be stored as a component of the local state. However, since it is frequently accessed, we store it separately as a function $pcf : Tid \to PC$, where PC is the enumeration type of all the available program counter labels.

The type $Action$ of actions of the automaton is the disjoint union of elements that are generated by default as follows:

- As required for proving linearizability, invoking and returning steps of nonatomic programs op (here: $op \in \{Enq, Deq\}$) have invoking and returning actions $inv_t(op, args)$, $res_t(op, vals)$, where $args$ are the inputs of the operations (if any), and $vals$ are the outputs[6].
- Deterministic steps of nonatomic programs are assigned the default action τ.
- Nondeterministic steps are assigned an action that fixes the nondeterministic choice. The programs of our case study have two nondeterministic steps: enqueue allocates an arbitrary reference $node \in Ref$ outside the heap, and dequeue allocates a new reference ref to a return value at line D2 in Algorithm 3. The latter gets the action $\texttt{chooseD2}(ref, oval)$ where ref is the chosen reference and $oval$ is the random initialization value. This allows a deterministic computation of the next state when the action is given.
- Atomic programs op like starting a thread t define an action $\texttt{call}_t(op, args, vals)$, global steps such as a flush define $\texttt{call}(op, args, vals)$ (with no subscript t on call).

6.3. The step relation of the automaton

The section gives the axioms defining the step relation $step \subseteq State \times Action \times State$ of the automaton which is automatically generated from the programs. To efficiently prove that individual steps of the programs satisfy certain properties, it is necessary to give a definition which ensures that most axioms and proof obligations can be defined *local* to one thread. This is done by using a precondition predicate together with three step functions for (1) global steps gstepf, (2) local steps lstepf and (3) the change of the program counter pcstepf.

$$\begin{aligned}
&\mathsf{pre} \subseteq GS \times LS \times PC \times Action\\
&\mathsf{gstepf} : GS \times LS \times PC \times Action \to GS\\
&\mathsf{lstepf} : GS \times LS \times PC \times Action \to LS\\
&\mathsf{pcstepf} : GS \times LS \times PC \times Action \to PC
\end{aligned}$$

These compute the next global and local state plus next program counter of a thread, given the previous ones and the action, provided the precondition holds. Based on these functions the step relation is defined by the axioms

$$\begin{aligned}
&\texttt{step}(\mathsf{mkstate}(gs, lsf, pcf), a, \mathsf{mkstate}(gs', lsf', pcf'))\\
&\leftrightarrow \exists\ t, ls', pc'.\ \texttt{lstep}(gs, lsf(t), pcf(t), a, gs', ls', pc') \wedge lsf' = lsf[t := ls'] \wedge pcf' = pcf[t := pc'] \qquad (3)\\
&\texttt{lstep}(gs, ls, pc, a, gs', ls', pc')\\
&\leftrightarrow \mathsf{pre}(gs, ls, pc, a) \wedge \mathsf{gstepf}(gs, ls, pc, a) = gs' \wedge \mathsf{lstepf}(gs, ls, pc, a) = ls' \wedge \mathsf{pcstepf}(gs, ls, pc, a) = pc' \qquad (4)
\end{aligned}$$

[6] Formally, the thread id t becomes an argument, but the name of the operation becomes part of the constructor name.

using an auxiliary predicate $\texttt{lstep} \subseteq GS \times LS \times PC \times Action \times GS \times LS \times PC$. The axioms as well as proof obligations given later use the convention that free variables are implicitly universally quantified.

The reduction to the three functions works for all the steps of nonatomic programs and for all atomic steps of threads which execute a single (parallel) assignment, e.g. those of IDQ [7].

Global steps like flushes and the crash are added as additional disjuncts to the formula defining the step relation (3). We demonstrate the translation with the crash, which is the most complex such step, as it is nondeterministic and modifies all local states. All other steps are deterministic and modify the global state only. Thus, they can be simplified to a predicate logic formula similar to the translation of atomic steps within a program given below.

The crash and the subsequent recovery are modeled as the program given in Algorithm 4. This program modifies gs, lsf and pcf. It first ramdomises all local variables (except the thread id of each thread) and the volatile global state (*vhead*, *vtail*, *vs*, *vRetRefs* and *vrvHp*) and sets $pcf(t)$ to crashed for all threads that have already started (are no longer in state idle). Then the recovery program is executed. This first restores the volatile heaps from the persistent ones. Then it restores the volatile head *vhead* by traversing the queue starting with *phead* until an unmarked cell is found. Finally it restores *vtail* by further traversing the queue until a null-reference is found. Together this gives the program in Algorithm 4.

It is easy to prove that this program always terminates, its step relation is added disjunctively to the previous definition (3) of the `step` predicate as one atomic step.

$$\begin{aligned}&\texttt{step}(\mathsf{mkstate}(gs, lsf, pcf), a, \mathsf{mkstate}(gs', lsf', pcf'))\\ \leftrightarrow\ &(a = \texttt{crash} \wedge \langle Crash() \rangle\ \mathsf{mkstate}(gs, lsf, pcf) = \mathsf{mkstate}(gs', lsf', pcf')) \vee (3)\end{aligned}$$

where `crash` is the action for a crash taken from the IDQ automaton, and program $Crash()$ is given in Algorithm 4. Formula $\langle p \rangle s = s'$ states that "there is a terminating run of the program p which starts in s and ends in state s'" in Dynamic Logic[8].

It finally remains to translate individual steps to axioms defining pre, lstepf, gstepf and pcstepf. The axioms are given for each program counter value $pc_1, pc_2, \ldots$ separately. We only give the axioms for the steps resulting out of a translation of line `D21`. Line `D21` is a persistence point for the dequeue operation (as marked by the with-clause) and hence sometimes becomes a non-τ action in the IO automaton. This is conditional on the current state of the queue: if the node to be dequeued still has its `deqID` to be -1, the node can be dequeued and hence the action is $do_{\mathrm{t}}(Deq)$. If this is not the case, the action is τ (because no dequeuing operation takes effect at this statement). This is specified in the precondition predicate:

$$\mathsf{pre}(gs, ls, \texttt{D21}, a) \leftrightarrow a = (gs.\texttt{ps}[ls.\texttt{nxt}].\texttt{deqID} = -1 \supset do_{ls.\texttt{tid}}(Deq);\ \tau)$$

This formula uses the shorthand notation $\phi \supset a_1;\ a_2$ for a conditional that computes a_1 if ϕ is true, and a_2 otherwise. Next, the three step functions are specified:

$$\begin{aligned}\mathsf{gstepf}(gs, ls, \texttt{D21}, a) &= gs.\texttt{ps} := gs.\texttt{ps}[ls.\texttt{nxt} := gs.\texttt{ps}[ls.\texttt{nxt}].\texttt{deqID} := gs.\texttt{vs}[ls.\texttt{nxt}].\texttt{deqID}]\\ \mathsf{lstepf}(gs, ls, \texttt{D21}, a) &= ls\\ \mathsf{pcstepf}(gs, ls, \texttt{D21}, a) &= \texttt{D22}\end{aligned}$$

The global step function describes the fact that step `D21` flushes the `deqID` field of the thread's local variable `nxt` from volatile to persistent memory. The notation employed for gstepf is KIV's notation for updating partial maps. The local state is left unchanged, and the program counter of a thread is moved from `D21` to `D22` by this step.

7. Thread-local proofs of assertions

A key ingredient for forward simulation proofs is the formalisation of an invariant that restricts the reachable states of the IOA of the implementation. Such an invariant typically comprises various assertions that are required to hold at different control points of the algorithm.

[7] Atomic steps, which execute programs more complex than just assignments are possible. They would be added disjunctively to the definition (4) of `lstep`. The case study does not use such steps.

[8] This is shorthand for $\neg\ wlp(p, \neg\ (s = s'))$ in weakest precondition calculus.

In earlier work [DDD+16], we have developed a thread-local proof method for establishing a global invariant for an IOA that has a structure consisting of mainly thread-local (program) steps. The proof method adapts traditional rely-guarantee approaches [Jon83, dRdBH+01] to our setting. For each case study, one can show that these proof obligations together guarantee a complex invariant, however, the definitions of the required predicates and associated lemmas are often tedious, and the proofs often time consuming. In this section, we define proof obligations that are automatically generated from the assertions given at program points. These enable one to generate the proof obligations and the definitions of the invariants from them automatically.

The proof system takes the following as input.

- For every label $pc_k \in PC$ an assertion φ_k that must hold for a thread whenever the thread is at the control point labelled pc_k. Within KIV, φ_k is encoded as a comment at label pc_k. Such assertions range over global variables and local variables of the thread in which the assertion appears. Recall that our program may contain special labels such as idle and ready; explicit assertions can also be given at such labels. Quite often, an assertion may range over a set of program locations (defined by program counter values). Explicitly given ranged assertions are conjoined to every φ_k where pc_k is in the range.
- A global invariant GInv*(gs)* over the global state gs.
- A rely predicate Rely*(gs, t, gs')*. All steps which are *not* executed by thread t should satisfy this predicate when they start in global state *gs* and end with *gs'*. Thread t *relies* on other threads to change the global state according to Rely.

For these assertions, three types of proof obligations are generated.

- For every step from label pc_i to pc_j with action a:
  `step-i-j`: $\varphi_i(gs, ls) \wedge \mathsf{GInv}(gs) \wedge \mathsf{pre}(gs, ls, pc_i, a) \rightarrow \varphi_j(\mathsf{gstepf}(gs, ls, pc_i, a), \mathsf{lstepf}(gs, ls, pc_i, a)) \wedge \mathsf{GInv}(\mathsf{gstepf}(gs, ls, pc_i, a))$
- For every step from label pc_i:
  `rely-i`: $\varphi_i(gs, ls) \wedge \mathsf{GInv}(gs) \wedge \mathsf{pre}(gs, ls, pc_i, a) \wedge ls.\mathtt{tid} \neq t \rightarrow \mathsf{Rely}(gs, t, \mathsf{gstepf}(gs, ls, pc_i, a))$
- For every label pc_i:
  `stable-i`: $\varphi_i(gs, ls) \wedge \mathsf{GInv}(gs) \wedge \mathsf{Rely}(gs, t, gs') \rightarrow \varphi_i(gs', ls)$

The first proof obligation (`step-i-j`) guarantees that each thread-local step guarantees the thread local assertion at the next control point (pc_j) and preserves the global invariant. The other two proof obligations ensure that steps of other threads do not invalidate assertions. This is split into showing that all such steps are rely steps (`rely-i`), and that all assertions are stable with respect to the rely (`stable-i`).

In the generated proof obligations, two simplifications are possible. First, since many steps do not update the global state ($\mathsf{gstepf}(gs, ls, pc_i, a) = gs$) the rely proof obligations can be dropped provided the Rely predicate is reflexive. Second, if φ_i and φ_j, for $i \neq j$, are syntactically the same formula, `stable-i` and `stable-j` are the same proof obligation, so only one of these is generated.

Moreover, for proofs of persistent memory programs, the steps corresonding to a flush only change the global state, thus their proof obligations are simpler. Thus, for these, the `step-i-j` proof obligation only requires one to prove that GInv(gs) is preserved, while the `rely-i` proof obligation requires the change to satisfy the Rely for all threads (since a flush is executed by the system). Therefore such steps preserve all assertions of all threads by the `stable-i` conditions and no extra stability proof obligations is required for the flushing step itself.

Within KIV, the proof obligations are defined such that a predicate Inv(gs, lsf, pcf) is invariant, where Inv is defined as follows:

$$\mathsf{LInv}(gs, ls, pc) \leftrightarrow \bigwedge_{pc_i \in PC} (pc = pc_i \rightarrow \varphi_i(gs, ls))$$

$$\mathsf{Inv}(gs, lsf, pcf) \leftrightarrow \mathsf{GInv}(gs) \wedge \forall\, t.\, lsf(t).\mathtt{tid} = t \wedge \mathsf{LInv}(gs, lsf(t), pcf(t))$$

The definition uses a local invariant LInv(gs, ls, pc) for each thread which is generated from the assertions and is a *very* large formula for the case study (several pages of text). Defining and maintaining this huge formula manually has been a severe problem in earlier case studies. Note, however, that for a specific label pc_i, the formula LInv(gs, ls, pc_i) reduces to the assertion $\varphi_i(gs, ls)$ given at that label which is usually between two and five lines of text.

It remains to define proof obligations for the global steps. For the crash

$\texttt{step-crash}: \mathsf{GInv}(gs) \wedge \mathsf{LInv}(gs, ls, pc) \rightarrow wp(Crash, \mathsf{GInv}(gs) \wedge \mathsf{LInv}(gs, ls, pc))$

has to be proven using weakest-precondition calculus. The other global steps are the flushes. These execute a single assignment that modifies global state only. For these the wp-formula simplifies to predicate logic. As an example, for the flush of $vhead$ with the assignment $phead := vhead$ the postcondition of the implication is $\mathsf{GInv}(gs.\texttt{phead:=}\ vhead) \wedge \mathsf{LInv}(gs.\texttt{phead:=}\ vhead, ls, pc)$.

The following theorem shows the soundness of this thread-local proof technique, i.e., it shows that the above described proof obligations are sufficient for proving invariants.

Theorem 7.1 If `step-i-j`, `rely-i`, `stable-i` and `step-crash` hold, and $\mathsf{Inv}(gs, lsf, pcf)$ holds in initial states, then Inv is an invariant of the automaton generated from the program, i.e.,

$\mathsf{Inv}(gs, lsf, pcf) \wedge \texttt{step}(gs, lsf, pcf, gs', lsf', pcf') \rightarrow \mathsf{Inv}(gs', lsf', pcf')$

Proof. The proof is by showing that `step-i-j`, `rely-i`, `stable-i` are sufficient to establish the standard decomposition lemmas

- `step-lemma`:
  $\mathsf{GInv}(gs) \wedge \mathsf{LInv}(gs, ls, pc) \wedge \texttt{lstep}(gs, ls, pc, a, gs', ls', pc')$
  $\rightarrow \mathsf{GInv}(gs') \wedge \mathsf{LInv}(gs', ls', pc') \wedge ls.\texttt{tid} = ls'.\texttt{tid}$
- `stable-lemma`:
  $\mathsf{GInv}(gs) \wedge \mathsf{LInv}(gs, ls, pc) \wedge \mathsf{GInv}(gs') \wedge \mathsf{Rely}(gs, ls.\texttt{tid}, gs') \rightarrow \mathsf{LInv}(gs', ls, pc)$
- `rely-lemma`:
  $ls.\texttt{tid} \neq t \wedge \mathsf{GInv}(gs) \wedge \mathsf{LInv}(gs, ls, pc) \wedge \texttt{lstep}(gs, ls, pc, a, gs', ls', pc')$
  $\rightarrow \mathsf{Rely}(gs, t, gs')$
- `otherstep-lemma`:
  $\mathsf{GInv}(gs) \wedge \mathsf{LInv}(gs, ls, pc) \wedge ls.\texttt{tid} \neq lsq.\texttt{tid} \wedge \mathsf{LInv}(gs, lsq, pcq) \wedge \texttt{lstep}(gs, ls, pc, a, gs', ls', pc')$
  $\rightarrow \mathsf{LInv}(gs', lsq, pcq)$

The first three lemmas follow from the proof obligations, by expanding the definition of `lstep` and by a case split over all possible pc_i. Lemma `otherstep-lemma` states, that the local invariant LInv of thread $lsq.\texttt{tid}$ is stable, when thread $ls.\texttt{tid}$ executes a step. It combines the previous thre lemmas: `step-lemma` and `rely-lemma` allow to infer $\mathsf{GInv}(gs')$ and $\mathsf{Rely}(gs, t, gs')$ from the preconditions of the implication, enabling `stable-lemma` (with lsq and pcq instantiating ls and pc) which gives the postcondition.

The proof of the theorem has to first split away the global steps from the definition of `step`.

- For global steps, the proof follows directly from lemmas like `step-crash` by instantiating ls and pc with any $lsf(t)$ and $pcf(t)$ that results from expanding the definition of Inv.
- Otherwise there is a thread t, such that the step is an `lstep`. By the definition of `lstep`, it has to be shown that $\mathsf{Inv}(gs', lsf[t := pc'], pcf[t := pc'])$ holds after the step is taken, when $\mathsf{Inv}(gs, lsf, pcf)$ and $\mathsf{pre}(gs, lsf(tid), pcf(t))$ holds and we have that for some action a:

  $$gs' = \mathsf{gstepf}(gs, lsf(tid), pcf(tid), a)$$
  $$ls' = \mathsf{lstepf}(gs, lsf(tid), pcf(tid), a)$$
  $$pc' = \mathsf{pcstepf}(gs, lsf(tid), pcf(tid), a)$$

  respectively.

Unfolding Inv, we have to prove that $\mathsf{GInv}(gs')$ and $\mathsf{LInv}(gs', ls', pc')$ hold, and that $\mathsf{LInv}(gs', lsf(t), pcf(t))$ is true for every $t \neq ls.\texttt{tid}$. The first two properties follow from `step-lemma`. Finally, `otherstep-lemma` gives the desired conclusion, by instantiating lsq an pcq with $lsf(t)$ and $pcf(t)$ respectively. □

8. Verification of the persistent queue

In this section, we apply our methodology to the persistent queue. In Section 8.1 we give an overview of the key properties needed for our proof. Section 8.2 describes the use of auxiliary variables that we use to track ownership

of queue nodes. Section 8.3 shows how persistence points are specified and explains the auxiliary (greyed out) code in Algorithm 3. In Section 8.4, we define our forward simulation relation. Section 8.5 briefly describes invariants and properties of the queue that are necessary for the proof, but not critical to the main ideas. Finally, Section 8.6 gives a summary of the effort needed to verify the case study.

8.1. Key properties of the queue data structure

There are several key properties that the persistent queue must maintain in order to ensure correctness. These properties are formalised as conjuncts of the global invariant of our proof.

Consider Fig. 4, which represents a state of the queue data structure. The first part of this structure, the *old reference list*, contains references to nodes that have been logically dequeued from the queue. The second part, the *queue reference list* contains references to nodes that have been enqueued but not yet dequeued. To track these lists, we introduce two auxiliary variables: orl for the old references and qrl for the queue references, which are disjoint lists over the Ref type. The expression $orl + qrl$ is the concatenation of our two lists, and we use the expresson qrl.`last` to mean the last element of qrl when qrl is nonempty. One problem that we must solve is to constrain the set of fields of nodes in the persistent queue that may be in a *volatile* state: that is, the unpersisted fields for which ps and vs disagree. Our invariants require that the `val` and `next` fields of each node referenced by an element of $orl + qrl$ are both nonvolatile, so that vs and ps agree on these fields, with the possible exception of the `next` field of last node in qrl. The enqueue operation ensures these fields are nonvolatile before a node is enqueued. We further require that every $deqID$ field of every node is nonvolatile, except possibly the $deqID$ of the first element qrl.`head` of qrl. We explain these exceptions below.

The references in the queues are ordered such that for any two references r, r' that are adjacent and in that order in $orl + qrl$, we have $ps[r]$.`next` $= r'$. Intuitively, the nodes referenced by elements of qrl contain the values that are currently in the queue and accordingly, we require that every such node has a persistent $deqID$ of -1 indicating that it has not been dequeued. We require that $vhead$ be either the last or second to last element of orl and that $phead$ is always an element of orl. A node is considered to be in the current queue state if it can be reached from $phead$ by following `next` fields, and its `deqID` field in persistent memory is -1. It is these properties that guarantee that the recovery procedure is always able to find the logical head of the queue by traversing next pointers from $phead$. Fig. 4 illustrates a state satisfying our invariant where $phead \neq vhead$.

Nodes move from qrl to orl during dequeue operations. The first step in this process is when a dequeuing thread successfully enters its id into the $deqID$ field of some node in line `D19` of Algorithm 3. It can be shown using the local assertions of the dequeue operation that this node is always the first node of qrl, and that this node is referenced by $ps[vhead]$.`next`. At this point, the effect of the CAS has not been persisted, and the $deqID$ field of $ps[vhead]$.`next` is thus volatile. In the original queue, the linearization point that removes the element from the queue is executed with the successful CAS. This is not possible here, as a crash after the CAS would yield a persistent queue where the element d is still in the queue, "undoing" the linearization. To have a one-to-one match with the abstract queue, we therefore leave orl and qrl unchanged on a successful CAS.

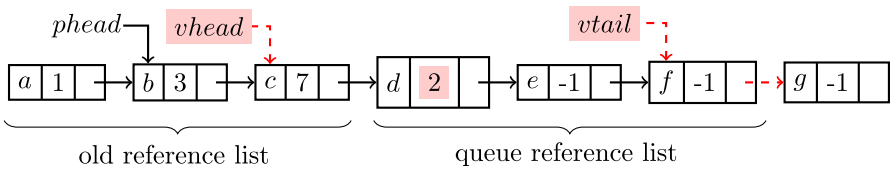
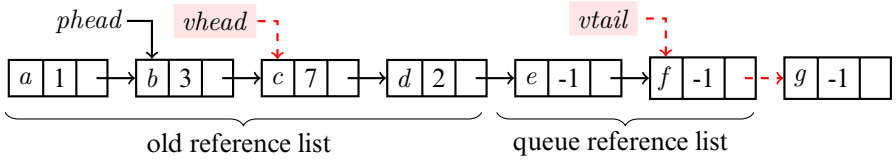


Fig. 4. Possible state of persistent queue; volatile data represented using shading and volatile pointers represented using dashed arrows

The next step in the process is that this field is flushed, either by the dequeuing thread at line `D21`, a helping thread, or the NVM system's flush action. This flush is the *persistence point* of the dequeue operation. Persistence points are analogous to linearization points in conventional linearizability [DD15]. Roughly speaking, the linearization point of an operation is the point when the effect of the operation becomes visible to all threads. This is also true of persistence points, with the additional requirement that once a persistence point has been reached,

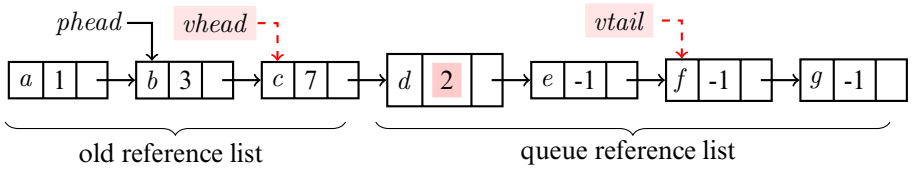
the effect of the operation must still be visible after a crash. Thus, an update to volatile memory that might be a linearization point in the conventional setting will typically not be a persistence point. Rather, the persistence point is often the moment when this update is flushed to the persistent store.

Line D21 moves qrl.head to the end of orl. The corresponding updates to qrl and orl in the auxiliary code of line D21 are explained in detail in the next Section 8.3. For the queue depicted in Fig. 4, the queue immediately after the volatile deqID of node d is flushed is as follows.



The abstract queue corresponding to this queue is $\langle e, f\rangle$. Note that in the queue above, $vhead$ pointer is now lagging (i.e. the pre-to-last element orl.butlast.last of orl) and must be updated to point to the new sentinel node d. This final step of the process of the process must occur, before another successful CAS of a dequeue is possible. Therefore a lagging *vhead* implies that the first element of qrl must still have $deqID$ field -1 in volatile store.

The process by which a node is added to qrl is simpler. Recall that the newly allocated node of an enqueue operation is added to the queue data structure by the CAS at line 9 in Algorithm 2. The persistence point of the enqueue operation is the point when the effect of this CAS is flushed (no later than line 10) and it is at this moment that the reference to the enqueued node is added to qrl. Consider again the queue in Fig. 4. The queue immediately after the next pointer of f becomes persistent is as follows.



Note that this transformation must be performed before moving $vtail$, otherwise the nodes after g could be lost upon system crash. In the queue above, $vtail$ is lagging and hence must be updated before a new node can be enqueued. As soon as the next pointer of f becomes persistent, the node g is considered to be part of the queue, i.e., the abstract queue corresponding to the queue above is $\langle d, e, f, g\rangle$.

8.2. Ownership-based properties

The invariants and rely conditions we use must be able to track exclusive read/write access of different shared memory components. To this end, we employ an ownership-based mechanism, which describes how threads are accessing different shared resources.

In our queue example, the shared resources are the node references, and the owners are threads that are executing the queue methods. In particular, we maintain an auxiliary variable $owns : Ref \rightarrow Ownership$, that maps each node reference to a value representing the ownership status of the node. We distinguish between two forms of ownership, *strong* and *weak ownership*. Both forms guarantee that whenever a thread owns a node reference, the fields of the node are not changed by other threads. Strong and weak ownership, however, have different stability properties. Strong ownership is *stable*: whenever a thread t strongly owns a reference, after any step of any other thread, thread t still strongly owns the reference in the post state. We model this property using a rely condition in Section 8.5. Weak ownership on the other hand is not stable, and may be removed by other threads. Note that during execution, a thread may transition from strongly owning a reference to weakly owning that reference. A thread may also give up ownership altogether.

Ownership and enqueues. Each enqueue operation is responsible for ensuring that key invariants of the queue nodes are satisfied when it adds its new node. To do this, we must ensure that newly allocated nodes are disjoint from the nodes of the queue.

When a thread t allocates a new node, $node$, at the start of enqueue, we set $owns(node) = Strong(t)$ to reflect the fact that t owns $node$. At that point no other thread can modify $node$, i.e., this is sufficient to guarantee that distinct threads modify distinct nodes, and thus that each thread can rely on the `next` and `val` fields of its own node not changing in volatile memory. After a successful CAS at line 9, ownership of $node$ by thread t changes to $Weak(t)$. Now t itself or another thread is able to execute a **flush**, which adds $node$ to the persistent queue. When this **flush** occurs, we set $owns(node) = None$.

Our invariant requires that each reference r in orl + qrl satisfies $owns(r) = None$, reflecting the fact that the set of nodes in the queue is disjoint from the set of nodes being initialised by some enqueue operation. To achieve this, it is sufficent to set $own(r) = None$ when r is added to qrl, which occurs when the next field of $vs[qrl.\texttt{last}]$ is flushed at a point when $r = vs[qrl.\texttt{last}].\texttt{next}$. Now, because every reference in orl + qrl has no owner but every modification of a `next` or `val` field occurs when the modifying thread has strong ownership, we can show that these modifications never change these fields within the queue data structure.

As we describe in Section 8.3, the persistence point of an enqueue operation is the moment when the reference to the operation's newly allocated node is flushed to the persistent store, which occurs after the CAS at line 9. We would like to use the ownership state of this reference to determine the id of the thread that enqueued the node. Thus we cannot set $owns(node) = None$ after the CAS at line 9, where $node$ is the local variable containing a reference to the new node. On the other hand, we cannot allow $owns(node) = Strong(t)$, because then t must "lose" this ownership when the field is flushed, contradicting the stablility of strong ownership.

We resolve this dilemma by using *weak ownership*, and setting $owns(node) = Weak(t)$ at the CAS at line 9. Our invariant requires that if the `next` field of $vs[qrl.\texttt{last}]$ is in a volatile state, then we have that $owns(vs[qrl.\texttt{last}].\texttt{next}) = Weak(t)$ for some thread t that has not yet passed its persistence point. As described in the next section, we then use this value to construct the abstract action to simulate at this step.

Helping and the return value array. As described in Section 2, the persistent queue employs a helping mechanism to allow dequeuing threads to complete another thread's concurrent dequeue operation (see lines 27 to 32). The key feature of the helping mechanism is that a helping thread writes the value from the dequeued node into the other thread's current return-value object, and flushes that object to the persistent store. The challenge presented by this part of the algorithm is in dealing with this thread-to-thread synchronisation in the context of a thread-local proof.

In our formal model, we specify that the dequeue procedure returns the local variable $optval$, and this persistent queue is durably linearizable. However, in order to enable the program to recover from a crash, the persistent queue ensures that this value is persistently available in $prvHp[vRetRefs(t)]$. To reflect this in our KIV development, we prove that the assertion $optval = prvHp[pRetRefs(t)]$ holds at the last line `D34` of the dequeue code. This is one of the most tricky parts of the verification and requires a new auxiliary variable and new invariants as we now explain.

We introduce a new auxiliary variable $rvOwns$ which maps each return-value object reference to a value describing the logical state of the reference. We update this auxiliary function as follows:

- When a thread t allocates a new return-value object r (line `D2`), $rvOwns(r)$ is set to $Pending(t)$.
- When a thread t claims a node to be dequeued (by successful execution of the CAS at line `D19`) $rvOwns$ $(vRetRefs(t))$ is set to $Claimed(t, v)$ where v is the value stored in the val field of the node claimed.

It is useful to think of this technique as an extension of the ownership technique that we use to manage queue nodes. We think of the thread argument of each $rvOwns$ value as the *owner* of the reference. Only steps of the owner thread are capable of changing an $rvOwns(r)$ value, and we encode this property in our rely condition. Furthermore, given this rely property, it is straightforward to show the following *return-value ownership* property: for all threads t, if $vRetRefs(t)$ is not null, then t is the thread id of $rvOwns(vRetRefs(t))$.

We now describe how the $rvOwns$ variable is designed to support the persistent queue's helping mechanism. After a thread t executes the CAS at line `D19`, installing its id in the $deqID$ field of the dequeued node nd, it is possible for another thread t' to attempt to help t by writing the dequeued value into the return-value object of t. When this happens, there is a race between the write of t at line `D22` and the helping write of t' at line `D29`, and the correctness of the helping mechanism depends on t' writing the correct value. Recall that $rvOwns(vRetRefs(t))$ is set to $Claimed(t, vs[nd].\texttt{val})$, when t successfully executes the CAS at line `D19`. We use the value in this $Claimed$ state to ensure that t' writes the correct value into the return-value object when it executes the *helping*

write at line D29. To achieve this, the precondition of the helping write of t' implies

$$rvOwns(addr) = Claimed(othert, vs[nd].\texttt{val}) \tag{5}$$

where $othert$ is a local variable of t' which records the value of t in this scenario. We explain the value of $addr$ shortly. Our rely relation is defined so that for any allocated reference r, if $rvOwns(r)$ is in the $Claimed$ state, then the value stored at that reference does not change, *and* that $vs[nd].\texttt{val}$ never changes for any allocated node. These two properties are sufficient to ensure the stability of (5).

Note that (5) is sufficient to prove that our helping mechanism is correct, so long as $addr$ points to the appropriate return-value object. That is, for every thread t'', if $addr = vRetRefs(t'')$ then $t'' = t$, but this follows from the return-value ownership property described above.

This concludes our overview of how we manage the persistent queue's helping mechanism. The reader is referred to the KIV development for the full treatment.

8.3. Identification of persistence points

Finding *persistence points* is similar to standard linearizability, where proofs proceed via identification of *linearization points* [DD15]. However, in durable linearizability, persistence points are typically statements (flush events) that cause the operation under consideration to become durable. Thus these statements must be simulated by the abstract do operation. Note that persistent points must occur after an operation has taken effect in NVM, but before the operation returns.

In MSQ, both the enqueue and the dequeue operation linearize upon successful execution of the CAS at line 9 of the enqueue and line 24 of the dequeue of the C code in Algorithm 2. However, in the persistent queue, these volatile memory actions *cannot* the persistence points of these operations, since their effects can be lost by an immediately subsequent crash. Rather, in the persistent queue, the persistence point is the first operation that *flushes the effect of this CAS*.

These flushes may occur by the same thread in the line following the CAS, or by another thread helping, or due to a system-controlled flush. Despite the actual persistence point being any of these possibilities, it is still possible to prove forward simulation with respect to the $do(Enq)$ operation of the intermediate automaton *IDQ*.

To enable an elegant refinement proof we use a feature of KIV that allows us to overwrite the default internal action τ of program steps given in Section 6.2 with a custom action. We overwrite the default action with the corresponding action of *IDQ* given in Fig 3, thus allowing the forward simulation to have a one-to-one match between actions. Technically, the KIV code of Algorithm 3 has **with** clauses after line numbers that specify the custom action.

For dequeue operations that return a value (rather than `empty`), the persistence point is the flush at line D21, if the *deqID* field has not yet been flushed already. Therefore, the action is set to the action $do_t(Deq)$ of the intermediate automaton *IDQ*, when the persistent *deqID* field $ps[nxt].\texttt{deqID}$ is still -1 (recall that $(\varphi \supset t_1;\ t_2)$ evaluates to t_1 when φ is true, and to t_2 otherwise). The dequeue operation can also help another thread $othert$ to execute the persistence point $do_{othert}(Deq)$ of its dequeue operation in line D28 under the same condition. Finally, the persistence point may also be a global flush of the $vs[r].\texttt{deqID}$ to $ps[r].\texttt{deqID}$, when r is $qrl.\texttt{head}$, the first element of the current queue references. In all cases the auxiliary variables qrl and orl that store the current and old reference list get updated by moving the head of qrl to orl.

The enqueue persistence points are similar. A successful CAS at line 9 appends a new node to the queue, but this CAS is not persisted until a subsequent flush, which is the enqueue operation's persistence point. For the C code of Algorithm 2, the flush may occur in line 10 of the same thread, line 14 executed by another thread, or with a global flush of a *next* field. An additional persistence point, where the action $do_t(Enq)$ is executed, is line D16 in the dequeue algorithm shown in Algorithm 3. Compared to dequeue, an additional problem is to determine the thread t that executes the persistence point, since the thread id is not stored in a field. The relevant thread is always the one that allocated the node that is now enqueued. The *next* field points to this node, but without inspecting all the references stored in the local $node$ variables, which contradicts thread-local reasoning, it is not possible to determine the thread. We use the ownership of the reference in the field being flushed to determine the identity of the relevant thread. For a node referred to by a reference nd that has been added to the queue but not yet flushed, our invariant guarantees that $owns(nd) = Weak(t)$ where t is the id of the thread that allocated

and added the node. We use this id to determine the action $do_t(Enq)$ to perform at the abstract level when the flush occurs.

Line D16 of Algorithm 3 provides an example of such a flush occurring. At line D16, the next reference of the cell that local variable *last* points to is flushed, therefore the **with** clause has action $do_{\text{owns(nxt)}}(Enq)$ if this reference is the one at the end of qrl. Note that it is possible that the reference has been flushed already, even that more nodes have been enqueued already. The check $last = (orl + qrl).\texttt{last}$ ensures together with the assertion $nxt = vs[last].\texttt{next} \wedge nxt \neq null$ which holds at D16 that the reference has not been flushed already, since all steps that execute the persistence point add the enqueued reference to the end of qrl as done by the auxilary code at D16.

Finally, the case of a dequeue that finds an empty queue, and returns value `empty`, has to be handled. The verification of the empty dequeue follows a similar pattern to the verification of the empty dequeue of MSQ. The persistence point is conditional on the future execution of the operation, thus we refer to the persistence point as a *potential persistence point* (this is similar to the concept of potential linearization points [DSW11, DGLM04, DD15]). The empty dequeue potentially takes effect at line D9, when the value loaded for nxt is `null`, but this decision is not resolved until later when the test at line D12 succeeds. Using the intermediate automaton (Fig. 3), allows the proof to proceed via forward simulation, like earlier proofs of linearizability [DD15, DGLM04], by executing action $checkEmp_t$, when the loaded reference nxt is `null`, and by executing $do_t(Deq)$ when exectution reaches line D14.

8.4. Forward simulation

The abstraction relation that is used in the forward simulation relates the state of the automaton generated from the programs to the intermediate automaton in Fig. 3. This can be split into a mapping between the global states, and two mappings `absls` and `abspc` between the local state and the label for each thread t.

$$\begin{aligned} & \texttt{abs}(\mathsf{mkstate}(gs, lsf, pcf), \mathsf{mkastate}(q, alsf, apcf)) \\ \leftrightarrow \quad & \texttt{content}(gs.\texttt{qrl}, gs.\texttt{ps}) = q \\ & \wedge \forall\ t.\ \texttt{abspc}(gs, lsf(t), pcf(t)) = apcf(t) \wedge \texttt{absls}(gs, lsf(t), pcf(t), alsf(t)) \end{aligned}$$

The global state of the intermediate automaton is the abstract queue q, so the global mapping just extracts the content of qrl by reading each value-field in the persistent store with a function $\texttt{content}(qrl, ps)$.

Relating the program labels to the abstract ones for one thread t is done with a function `abspc`. For the three labels idle, ready and crashed that are common to both automata the function is identity.

All program labels in the dequeue program that are before the persistence points are mapped to the abstract label *inv(Deq)*, while those after the persistence points are mapped to *ret(Deq)*. In most cases the decision depends on the concrete program counter only. Labels that are both before the successful CAS at line D19 in Algorithm 3 as well as before the persistence point of an empty dequeue at D14 are mapped to *inv(Deq)*, while lines after D14 and after the flush at D21 are mapped to *ret(Deq)*. There are three lines in the code where the concrete label alone is not sufficient: at lines D20, D21 the CAS must have been successful, and the *next* reference must have already been flushed, for the thread to be in state *ret(Deq)*. The label D5 at the start of the while loop is mapped to *ret(Deq)*, if its test is already false. The enqueue program is similar, it just does not have to consider the special case of returning empty.

It remains to define a relation `absls` between the concrete local state ls and the abstract local state als of a thread. This relation ensures that the input values for enqueue that were set on invoke on both levels are identical before the persistence points, and that the output values of dequeue after the persistence point are identical too. For the abstract level the output value is $out(t)$, for the concrete level there are three cases. After a successful CAS of dequeue, the output value is first stored in $vs[nxt].\texttt{val}$. After setting the return value at line D22 it is in $vrvHp[vRetRefs(t)]$. The relation therefore needs to have the global state and the local program counter as parameters.

Finally, the local value $obsEmp(t)$ of the abstract level is required to be true, after a reference has been loaded into the local nxt variable at D9, when action $CheckEmp_t$ is executed. It must stay true until the persistence point at line D14.

8.5. Supporting invariants and rely properties

We have outlined the main points in our proof, but as is typical for verifications of this type, we require a number of supporting properties to complete the proof. The reference lists orl and qrl satsify the following additonal invariants:

1. All nodes in the old reference list must have been marked, i.e. have a `deqID` field different from -1 in both ps and vs, indicating that they have been dequeued.
2. All nodes in the queue reference list must have a *deqID* field value -1 in ps.
3. Only the first node in qrl may have a *deqID* field value other than -1 in vs. This results from a dequeue that has executed its CAS, but not yet flushed.
4. $vhead$ is either the last element of orl or it is lagging and is the prior to last element. In this case however, the head of qrl cannot be marked in vs. This ensures that a lagging head is "repaired" first before another dequeue can happen.
5. The *vtail*-pointer points to (orl + qrl).`last` or is lagging too, pointing (orl + qrl).`butlast.last`. In the latter case, ps[(orl + qrl).`last`].`next` must be `null`(no red reference at the end of the queue). Again, this ensures, that a lagging tail is repaired first, before another enqueue has a chance to a successful CAS.
6. In any case, $phead$ is never reachable from *vhead* and *vhead* is never reachable from *vtail*.

To maintain these invariants there are number of rely properties that enable thread-local reasoning. They ensure that steps of other threads than t will not destroy relevant properties:

1. the pointers *vhead*, *vhead* and *vtail* always move forward. Their new value is always reachable from the old value by following the chain of references in $orl + qrl$.
2. The lists qrl and orl also move forward: the list orl and $orl + qrl$ before the steps of other threads are always a prefix of the lists after steps of other threads. This ensures that a local variable like $first$, which is initally set to $vhead \in orl$ still points to a reference in orl.
3. For both vs and ps, the $deqID$ field of the cells in $orl + qrl$ is always unchanged, or changes from unmarked to marked.
4. If for all references r in a suffix of $orl + qrl$, the $deqID$ field $vs[r]$.`deqID` is different from t, then this will be preserved by steps of other threads. This property is crucial to make sure that a thread helping with a dequeue will never compute its own threadid t as the value of $other t$ at line `D25`. Note that this property holds for the volatile store only, since another thread helping t in a dequeue may flush a reference with persistent $deqID$ field being t.

Again, the reader is referred to the KIV development for the full treatment.

8.6. Mechanization in KIV

The refinement has been mechanically proven in the interactive theorem prover KIV [EPS+15], which has been used extensively in the verification of concurrent data structures (e.g., [SDW14, DSW11]). See [KIV20] for the full KIV proofs and the encodings.

The proof consists of two parts. In the first part an invariant Inv(cs) was proved for the automaton C generated from the programs in Section 5 and the proof obligations resulting from it were discharged.

For the second part a mechanised theory for IOA that is available in KIV has been instantiated. This theory formalises a lot of the theory given in [LV95]. In particular, a theorem that forward simulation implies refinement is proven there. Instantiating this theorem gives the standard proof obligations of a forward simulation.

For our case study the specialised proof obligations of Definition 4.2 are used, where the concrete automaton C is the one generated from the programs, and A is IDQ. Reachability of concrete states ($cs \in reach(C)$) is approximated with the invariant $inv(cs)$. No invariant is needed for IDQ. Since we have made sure that even internal actions like $do_t(op)$ or $checkEmp_t$ are the same in both automata, the proof obligation for internal step correspondence proves the first disjunct for steps with action τ, and the second disjunct with abstract action chosen to be equal to the concrete action otherwise.

The main complexity of the proof is in proving the invariant with the thread-local proof technique of Section 7. Getting all the assertions and the rely conditions correct, and strong enough to imply critical lemmas about persistence points and return values required a lot of iterations. The final proofs for this part require ca. 4300 user interactions. Proving the step correspondence for forward simulation is less complex and took only a few iterations (which mainly lead to a few additional requirements for the invariant). The final simulation proofs have ca. 1200 user interactions.

9. Conclusions and related work

The recent development of NVM has been accompanied by persistent versions of well-known concurrent constructs, including concurrent objects [FHMP18, CAL18], synchronisation primitives [HPM+18, PKMH18] and transactional memory [JNCV18]. Developing concurrent objects implemented for NVM presents a similar challenge to weak memory, in the sense that there are multiples levels of memory to consider. Moreover, caches and registers are volatile, while cache flush instructions allow reordering with store instructions in accordance with the memory model of the system (e.g., [RV18]). Correctness in the presence of crashes and recovery can be affected by the order in which elements are persisted, which necessitates the use of programmer-controlled flush operations, increasing complexity. This paper has focussed on a persistent queue [FHMP18], against the recently developed notion of durable linearizability [IMS16], extending the prior our work [DDD+19].

Research on verification and validation under persistent memory is still in its infancy. Recent works have presented techniques for testing persistent memory applications [LWZ+19, OBLL16], including works on testing linearizability under persistence [CCL+19]. Denny et al [DLV16] have considered static analysis techniques for verifying correctness language extensions to C, including extensions that support atomic transactions. However, these works do not cover full verification, or durable linearizability. A model checking approach to verifying the Friedman et al queue (i.e., the same case study as our paper) has also been developed [IU18]. On the one hand, the authors address the TSO memory model, but on the other hand, the verification is incomplete. Although no errors were found, the authors state: "This result cannot guarantee correctness of the queue because of many reasons. For example, it was bounded model checking, user threads were assumed to work in accordance with a fixed scenario, the checked property did not cover the entire durable linearisability, and the model was unclear to be implemented correctly." Developing simulation-based techniques to address correctness of concurrent data structures in the persistent-TSO model [RWNV20, RV18] remains future work.

There are several works on full verification of (sequential) file systems that are tolerant to certain types of system crashes, e.g., using ASM refinement [EPSR16], crash-aware Hoare logic [CZC+15] and SMT-based reasoning [SBTW16]. More recently, verification techniques for concurrent systems with crashes have been developed [CTKZ19]. These techniques are currently used to verify high-level client applications, as opposed to fine-grained concurrent data structures, and hence, have not been applied to verify durable linearizability specifically. In other work, we have considered correctness of software transactional memory algorithms under persistent memory semantics via a new condition *durable opacity* [BDD+20] that extends opacity [GK10] to NVMs in the same way that durable linearizability extends linearizability.

Verification of durable linearizability for fine-grained persistent memory algorithms is inherently more complex than linearizability in the standard setting [HW90, DD15]. Since an operation only takes effect after a flush event, helping is inevitably required to bring the data structure into a consistent state and for an operation to take effect. For proofs by refinement, these additional helping steps have to be considered in the simulation proof. This ultimately complicates the invariants used since helping is performed by another thread or by the system. Moreover, since the state of the data structure can be "lagging" immediately after helping is performed, precisely formalising the underlying helping mechanism further complicates the invariant.

In summary, on the theoretical front we have introduced an operational characterisation of durable linearizability that can serve as the basis for mechanisable proofs of correctness. On the practical side, we have presented a general technique for performing such proofs using the KIV proof assistent. To use this technique on a particular algorithm, one first models the algorithm's shared variables and defines the actions that flush them, using a simple pattern. Then one writes the algorithm in KIV's programming language and annotates the program with persistence points (using KIV's **with** syntax). Then one applies KIV's automatic translation tool to obtain an automaton modelling the program with its flushes and crashes. Finally, one uses KIV's thread-local proof support to verify appropriate invariants and a simulation relation. We have exemplified this technique

using the persistent queue as a case study, and mechanically completed the verification in the interactive prover KIV.

This paper extends the results of our earlier version [DDD+19] in three ways. First, the automated translation from KIV programs to automata that can model flushes and crashes is novel. Second, in previous work, we only showed soundness of the durable automaton, whereas here we also demonstrate completeness. Third, we present a much more detailed discussion of the proof than previously.

Acknowledgements

We thank Lindsay Groves for comments that have helped improve an earlier version of this paper. Alexander Lindermayr and Kilian Kotelewsky helped to implement the framework for IOA in KIV while doing their Bachelor theses. Derrick, Dongol and Doherty are supported by EPSRC Grants EP/R032351/1, EP/R032556/1, EP/R019045/2; Wehrheim by DFG Grant WE 2290/12-1, Schellhorn by RE 828/13-2.

References

[BDD+20] Bila E, Doherty S, Dongol B, Derrick J, Schellhorn G, Wehrheim H (2020) Defining and verifying durable opacity: correctness for persistent software transactional memory. In: Gotsman A, Sokolova A (eds) FORTE, vol 12136 of LNCS. Springer, pp 39–58.

[CAL18] Cohen N, Aksun DT, Larus JR (2018) Object-oriented recovery for non-volatile memory. PACMPL 2(OOPSLA):153:1–153:22

[CCL+19] Cepeda D, Chowdhury S, Li N, Lopez R, Wang X, Golab W (2019) Toward linearizability testing for multi-word persistent synchronization primitives. In: Felber P, Friedman R, Gilbert S, Miller A (eds) OPODIS, vol 153 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp 19:1–19:17

[CTKZ19] Chajed T, Tassarotti J, Kaashoek MF, Zeldovich N (2019) Verifying concurrent, crash-safe systems with perennial. In: Brecht T, Williamson C (eds) SOSP. ACM, pp 243–258

[CZC+15] Chen H, Ziegler D, Chajed T, Chlipala A, Kaashoek MF, Zeldovich N (2015) Using crash hoare logic for certifying the FSCQ file system. In: Miller EL, Hand S (eds) SOSP. ACM, pp 18–37

[DD15] Dongol B, Derrick J (2015) Verifying linearisability: a comparative survey. ACM Comput Surv 48(2):19:1–19:43

[DDD+16] Doherty S, Dongol B, Derrick J, Schellhorn G, Wehrheim H (2016) Proving opacity of a pessimistic STM. In: OPODIS, vol 70 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp 35:1–35:17

[DDD+19] Derrick J, Doherty S, Dongol B, Schellhorn G, Wehrheim H (2019) Verifying correctness of persistent concurrent data structures. In: ter Beek MH, McIver A, Oliveira JN (eds) Formal methods—the next 30 years–third world congress, FM 2019, Porto, Portugal, October 7–11, 2019, Proceedings, vol 11800 of Lecture notes in computer science. Springer, pp 179–195

[DGLM04] Doherty S, Groves L, Luchangco V, Moir M (2004) Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig D, Núñez M (eds) Formal techniques for networked and distributed systems—FORTE 2004. Springer, Berlin, pp 97–114

[DLV16] Denny JE, Lee S, Vetter JS (2016) NVL-C: static analysis techniques for efficient, correct programming of non-volatile main memory systems. In: Nakashima H, Taura K, Lange J (eds) HPDC. ACM, pp 125–136

[dRdBH+01] de Roever WP, de Boer FS, Hannemann U, Hooman J, Lakhnech Y, Poel M, Zwiers J (2001) Concurrency verification: introduction to compositional and noncompositional methods, vol. 54 of Cambridge tracts in theoretical computer science. Cambridge University Press

[DSW11] Derrick J, Schellhorn G, Wehrheim H (2011) Verifying linearisability with potential linearisation points. In: Butler MJ, Schulte W (eds) FM 2011, vol 6664 of Lecture notes in computer science. Springer, pp 323–337

[EPS+15] Ernst G, Pfähler J, Schellhorn G, Haneberg D, Reif W KIV—overview and verifythis competition. Softw Tools Technol Transf STTT) 17(6):677–694, 2015

[EPSR16] Ernst G, Pfähler J, Schellhorn G, Reif W (2016) Modular, crash-safe refinement for asms with submachines. Sci Comput Program 131:3–21

[FHMP18] Friedman M, Herlihy M, Marathe VJ, Petrank E (2018) A persistent lock-free queue for non-volatile memory. In: Krall A, Gross TR (eds) ACM SIGPLAN symposium on principles and practice of parallel programming, PPoPP. ACM, pp 28–40

[GK10] Guerraoui R, Kapalka M (2010) Principles of transactional memory. Synthesis lectures on distributed computing theory. Morgan & Claypool Publishers

[HPM+18] Huang Y, Pavlovic M, Marathe VJ, Seltzer M, Harris T, Byan S (2018) Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In USENIX annual technical conference. USENIX Association, pp 967–979

[HW90] Herlihy M, Wing JM (1990) Linearizability: a correctness condition for concurrent objects. ACM TOPLAS 12(3):463–492

[IMS16] Izraelevitz J, Mendes H, Scott ML (2016) Linearizability of persistent memory objects under a full-system-crash failure model. In: Gavoille C, Ilcinkas D (eds) Distributed computing—30th international symposium, DISC, vol 9888 of Lecture notes in computer science. Springer, pp 313–327

[IU18] Iiboshi H, Ugawa T (2018) Towards model checking library for persistent data structures. In: IEEE 7th non-volatile memory systems and applications symposium, NVMSA 2018, Hakodate, Sapporo, Japan, August 28–31, 2018. IEEE, pp 119–120

[JNCV18] Joshi A, Nagarajan V, Cintra M, Viglas S (2018) DHTM: durable hardware transactional memory. In: ISCA. IEEE Computer Society, pp 452–465

[Jon83] Jones CB (1983) Tentative steps toward a development method for interfering programs. ACM Trans Program Lang Syst 5(4):596–619

[KIV20] KIV proofs for the durable linearizable queue, 2020. https://kiv.isse.de/projects/Durable-Queue.html

[LT87] Lynch NA, Tuttle MR (1987) Hierarchical correctness proofs for distributed algorithms. In: PODC. ACM, New York, pp 137–151.

[LV95] Lynch N, Vaandrager F (1995) Forward and backward simulations part I: untimed systems. Inf Comput Inf Control IANDC 121:214–233

[LWZ+19] Liu S, Wei Y, Zhao J, Kolli A, Khan SM (2019) Pmtest: a fast and flexible testing framework for persistent memory programs. In: Bahar I, Herlihy M, Witchel E, Lebeck AR (eds) ASPLOS. ACM, pp 411–425

[MS96] Michael MM, Scott ML (1996) Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings 15th ACM symposium on principles of distributed computing, pp 267–275

[OBLL16] Oukid I, Booss D, Lespinasse A, Lehner W (2016) On testing persistent-memory-based software. In: DaMoN. ACM, pp 5:1–5:7

[PKMH18] Pavlovic M, Kogan A, Marathe VJ, Harris T (2018) Brief announcement: persistent multi-word compare-and-swap. In: PODC. ACM, pp 37–39

[RV18] Raad A, Vafeiadis V (2018) Persistence semantics for weak memory: integrating epoch persistency with the TSO memory model. PACMPL, 2(OOPSLA):137:1–137:27

[RWNV20] Raad A, Wickerson J, Neiger G, Vafeiadis V (2020) Persistency semantics of the intel-x86 architecture. Proc ACM Program Lang 4(POPL):11:1–11:31

[SBTW16] Sigurbjarnarson H, Bornholt J, Torlak E, Wang X (2016) Push-button verification of file systems via crash refinement. In: Keeton K, Roscoe T (eds) OSDI. USENIX Association, pp 1–16

[SDW14] Schellhorn G, Derrick J, Wehrheim H (2014) A sound and complete proof technique for linearizability of concurrent data structures. ACM Trans Comput Log 15(4):31:1–31:37

[TK05] Tuch H, Klein G (2005) A unified memory model for pointers. In: Sutcliffe G, Voronkov A (eds) Logic for programming, artificial intelligence, and reasoning. Springer, Berlin, pp 474–488

Received 3 April 2020

Accepted in revised form 7 February 2021 by Annabelle McIver, Maurice ter Beek, Cliff Jones and Jim Woodcock