Spring 5-2022

A Machine-Verified Proof of Linearizability for a Queue Algorithm

Ugur Yavuz
Ugur.Y.Yavuz.GR@Dartmouth.edu

A MACHINE-VERIFIED PROOF OF LINEARIZABILITY FOR A QUEUE ALGORITHM

A Thesis
Submitted to the Faculty
in partial fulfillment of the requirements for the
degree of

Master of Science

in

Computer Science

by Ugur Y. Yavuz

Guarini School of Graduate and Advanced Studies
Dartmouth College
Hanover, New Hampshire

May 2022

Examining Committee:

_____
Prasad Jayanti (chair)

_____
Sergey Bratus

_____
Stephan Merz

_____
F. Jon Kull, Ph.D.
Dean of the Guarini School of Graduate and Advanced Studies

Abstract

Proofs of linearizability are typically intricate and lengthy, and readers may find it difficult to verify their correctness. We present a unique technique for producing proofs of linearizability that are fully verifiable by a mechanical proof system, thereby eliminating the need for any manual verification. Specifically, we reduce the burden of proving linearizable object implementations correct to the proof of a particular invariant whose correctness can be shown inductively. Noting that the latter is a task that many proof systems (such as the TLA+ Proof System we chose to work with) are well-suited to handle, this technique allows us to shift the responsibility of verification away from the reader and onto a machine, by enabling us to produce mechanically verifiable proofs of linearizability. We then demonstrate the effectiveness of this technique, which heretofore had only been applied to problems of a smaller scale, by proving the linearizability of a well-known queue algorithm whose proof of correctness is known to be challenging.

Preface

First and foremost, I would like to offer my most sincere gratitude and appreciation to my advisor, Dr. Prasad Jayanti, for his guidance and steadfast encouragement all through my thesis project, and for inspiring me to pursue higher studies. I would also like to thank Dr. Sergey Bratus and Dr. Stephan Merz for their feedback and for agreeing to serve on my thesis examination committee. I am also thankful to Lizzie Hernández Videa and Siddhartha Jayanti, whose help and suggestions were indispensable for the completion of this project. Finally, I would like to thank my family; especially my parents Belgin and Hüdaverdi Yavuz, my aunt Nilgün Eroğlu Maktav and my grandmother Halide Eroğlu, for their unwavering support throughout my studies, and my friends at Dartmouth and back home in Turkey for their stimulating company and support.

Contents

List of Figures

1 Introduction

Our study focuses on concurrent, multiprocess computing systems. These systems are comprised of processes that each have unique identifiers and that are *asynchronous*, meaning that their relative execution speeds fluctuate arbitrarily over time. These processes communicate through the use of *typed* and *shared objects* that are stored in the system's memory. The *type* of a shared object determines the set of operations it supports, as well as the arguments those operations can be invoked with by individual processes; it also specifies how these operations affect the object's *state* and what response is returned to the invoking process after the execution of the operation is completed. A shared object is *atomic* if it is the case that whenever any process calls an operation on the object, the operation takes immediate effect; that is, the state of the object changes, and the invoking process receives a corresponding response instantaneously.

Microprocessors compatible with the well-known and widely used Intel® 64 and IA-32 architectures, for instance, can atomically execute operations such as *read*, *write*, *compare-and-swap* (also known as *compare-and-exchange*), and *fetch-and-add* on memory words, among many others [Int22]. When designing programs for concurrent computing systems, however, there is frequently a need for types of atomic objects that are often not supported by the hardware of the system by default and that embody useful abstract data structures such as hash tables, snapshots, and queues. This gap can be bridged by designing *algorithms* that implement atomic objects of the desired types, using atomic objects that are readily available.

An *algorithmic implementation* of an object specifies a method for each operation supported by the object. For example, an implementation of a *stack* [Cor+09] S provides the set of instructions for the methods $push_p(e)$ and $pop_p()$, that a process p can invoke to push an element e onto S, or pop an element from the top of S which is promptly returned to p as a response. Alternatively, an implementation of a *snapshot object* [Afe+93] S specifies the methods $write_p(i, v)$ and $scan_p()$ for a process p, which it can invoke to respectively write v in the i^{th} component of S, and to retrieve the values stored in all the components of S altogether.

It is important to note that when a process p invokes a method, the execution of the method lasts over a period of time determined by the length of the code composing the instructions for the method, the rate at which p executes code, as well as the interruptions by other processes executing their own instructions. Therefore, the operations of a process on an implemented shared object are not instantaneous; i.e., they are not inherently atomic. In a seminal publication in 1990, however, Herlihy and Wing [HW90] identify and rigorously define a property named *linearizability*, which, once satisfied by an implementation, allows an implemented object to behave identically to an atomic one. In essence, this property requires that each method appears to take effect instantaneously at some point (referred to as the *linearization point*) within the interval of time where the operations that implement the method are executed. The identification of this property was a significant milestone for the field of concurrent algorithms, to the extent that for this and subsequent work [Her91], Herlihy was awarded the 2003 Dijkstra Prize.

As expected, linearizable implementations of objects are highly desirable, as they are qualitatively indistinguishable from atomic objects. It is worth noting, however, that algorithms that constitute linearizable implementations are typically intricate, and they are prone to subtle yet potentially critical race conditions. As a

result, any linearizable algorithm must be accompanied by a robust proof of its linearizability. This presents a challenge, as such proofs are typically rather lengthy, and therefore the risk of an error being present in the proof is not insignificant, possibly masking faults in the suggested algorithm. The length of these proofs, together with the level of detail they entail and the level of abstraction the reader should process in order to grasp such proofs, make their rigorous verification impractical, if not impossible in certain situations.

An effective way out of this dilemma is to produce a proof of linearizability that is completely machine verified, or one that is mainly machine verified with only a tiny component requiring human verification. If such a component exists, it should ideally be simple to verify: so simple that the human verifier can confirm its correctness with as little knowledge of the algorithm and the underlying proof as possible. As part of an unpublished undergraduate research project, we have previously shown this goal to be, in fact, achievable. In essence, our claim arises from a theorem that asserts that for every algorithm A, A is linearizable if and only if a particular statement S is an invariant of a *transformation* A' of A. The reduction of the problem of determining whether A is linearizable to the problem of determining whether S is an invariant of A' is particularly useful because the elaborate definition of linearizability makes it challenging to produce proofs that can be expressed in a way that machine verifiers can handle.

As part of our previous work, we have successfully demonstrated that one can in fact use proof assistants to write a mechanically verifiable proof (that is, one that can be verified by an interactive theorem prover on a machine) that S is an invariant of A'. Such a machine-verified proof then implies the linearizability of A given the theorem we have established. Furthermore, a reader who wants to verify the correctness of this assertion, simply needs to examine if A' is indeed a

valid transformation of A; that is, whether A' can be derived from A using only certain transformation rules permitted by our theorem. As this examination is merely a basic syntactic check, human verification is quick and straightforward as desired.

Our technique consists of first determining a valid and useful transformation A' of A, given an algorithm A; identifying the invariants of A' (specifically one that implies the particular statement we are interested in, and that is strong enough to be proven inductively); then writing the specification for A' as well as its invariants in the TLA$^+$ specification language [Lam02; Mer08]; and finally writing a proof for the inductive invariance of the claimed invariant using, and verified by, the TLA$^+$ Proof System (TLAPS) [Cha+08]. This machine-verified proof, in turn, implies the linearizability of the original algorithm. We note that the choice of TLA$^+$ and TLA$^+$ Proof System is immaterial, but we found these tools to be well-suited for the aims of our project.

Until previously, this technique's uses had been limited to relatively small-scale problems (particularly, a simplified version of a snapshot algorithm [Jay05]). The Herlihy-Wing queue algorithm, described in the seminal linearizability paper [HW90], appeared to be a good candidate for the application of this technique, as a non-trivial algorithm whose proof is lengthy (consuming a significant portion of the paper and extending into the appendix) and requires careful reading to comprehend and verify. We therefore closely studied the Herlihy-Wing queue algorithm, and we successfully machine-verifiably proved that the algorithm is linearizable, with the use of our technique in conjunction with TLAPS.

2 The computational model

Before articulating our theorem, we must first provide rigorous definitions for the components of our computational model, a concurrent computing system, and for the concept of *linearizability*.

In a nutshell, *processes* and *shared objects* are the fundamental building blocks of *concurrent algorithms*; that is, algorithms that run on concurrent computing systems. An *implementation* is a particular type of algorithm, and a *linearizable implementation* is one that satisfies a particular property. We will offer formal definitions for each of these notions in this section, starting with the most fundamental ones.

Definition 2.1 (Processes and local variables)**.** Every *process* in a concurrent computing system has a unique name p, and is assigned a program Π_p to execute. A program consists of lines of code, each identified by its number, referring to where they appear in the program. Every line of the program of a process is made of one or possibly multiple operations. Within this sequence of operations, p may invoke one operation on a single shared object and receive a response, and read and modify a number of *local variables*.

Although any process can interact with the shared objects stored in the memory of a concurrent computing system, the local variables of a process p are only accessible to p itself, which explains the lack of a limit on the number of operations p can call on them as part of a line in its program. The *program counter* of p, de-

noted pc_p, is a specialized local variable that points to the number of a line in Π_p, and this line should be understood to be the line of Π_p that p will run next. The *state* (or more specifically, the *local state*) of p refers to the set of values assigned to the local variables of p.

As discussed in the introduction, the processes of a concurrent computing system communicate via *typed shared objects*.

Definition 2.2 (Objects and types)**.** An *atomic shared object* (which will, from this point onward, be simply referred to as an *object* in this paper) has a name O, and a *type* T. Types are defined as tuples $(\Sigma, \mathcal{P}, Op, Op(p), Arg(\alpha), Res(\alpha), \delta)$ whose components determine the following properties for any object O of type T:

- Σ is the set of allowed states for the object. The state of O (denoted $O.state$), consists of the object's properties and their present values, and at all times is an element of Σ.

- $\mathcal{P}$ is the set of processes that can interact with O. In other words, p can invoke operations on O if and only if $p \in O.\mathcal{P}$.

- Op is the set of *operations* supported by the object O. An operation $\alpha \in Op$ can be denoted as $O.\alpha$.
  $Op(p)$, on the other hand, is a function that maps processes to the set of operations that they are allowed to invoke on O. More precisely, p can invoke an operation α on O if and only if $p \in O.\mathcal{P}$ and $\alpha \in O.Op(p)$.

- $Arg(\alpha)$ is a function that maps an operation α in $O.Op$ to the set of arguments it can be invoked on O with (wherein $\perp$ denotes the empty argument, to be used for invocations with no arguments).

– $Res(\alpha)$ is a function that maps an operation α in $O.Op$ to the set of responses invoking processes can receive after the execution of the invoked method α on O. If no particular response is returned, we traditionally denote the response to be `ack`, which stands for the "acknowledgment" of the completion of the operation.

– δ is a partial function, with the following domain and codomain definitions:

$$\delta : O.\Sigma \times O.\mathcal{P} \times O.Op \times \bigcup_{\alpha \in O.Op} O.Arg(\alpha) \rightarrow O.\Sigma \times \bigcup_{\alpha \in O.Op} O.Res(\alpha),$$

with $\delta(\sigma, p, \alpha, \lambda)$, being defined for $\sigma \in O.\Sigma$, $p \in O.\mathcal{P}$, $\alpha \in O.Op(p)$ and $\lambda \in O.Arg(\alpha)$, as (σ', ρ) for some $\sigma' \in O.\Sigma$ and $\rho \in O.Res(\alpha)$, if and only if the invocation of α with the argument λ by a process p (who is allowed to invoke α, which can, in turn, be invoked with the argument λ) changes the state of O to σ' and returns ρ as a response to p.

We will exemplify this elaborate definition by providing the definitions of some familiar object types in this notation.

Example 2.1 (Read/write object)**.** A *read/write* object S supports an operation named *read*, which takes no arguments and returns $S.state$ to the invoking process; and an operation named *write*, which admits an argument v that is to replace the contents of $S.state$.

There are no restrictions on $S.\mathcal{P}$ and $S.Op$, meaning that every process can freely invoke either operation on S; and the state of the object can be anything that can be stored in single memory words of the computing system (e.g. $\mathbb{N}$, $\mathbb{Z}$, or Φ^* for some finite alphabet of characters Φ). In view of these properties, the *read/write object* type would be defined as the tuple $S := (\Sigma, \mathcal{P}, Op, Op(p), Arg(\alpha), Res(\alpha), \delta))$

where:

- $Op := \{read, write\}$ and for any $p \in S.\mathcal{P}$, $Op(p) := \{read, write\}$.
- $Arg(read) := \emptyset$ and $Arg(write) := S.\Sigma$.
- $Res(read) := S.\Sigma$ and $Res(write) := \emptyset$.
- $\delta(\sigma, p, read, \perp) := (\sigma, \sigma)$ and $\delta(\sigma, p, write, v) := (v, \texttt{ack})$, for $\sigma \in S.\Sigma$, $p \in S.\mathcal{P}$ and $v \in S.Arg(write)$, recalling that $\perp$ stands for the empty argument, and that `ack` is the standard response for operations that do not have a particular return value.

Example 2.2 (Queue)**.** A *queue* object Q embodies a sequence, and supports an operation named *enqueue*, which admits an argument v that is to be placed at the end of the sequence; and an operation named *dequeue* which takes no arguments, and if the sequence is not empty, returns the first element to the invoking process, and removes it from the sequence.

There are no restrictions on $Q.\mathcal{P}$ and $Q.Op$, meaning that every process can invoke either operation on Q (with the caveat that the operation is left undefined for when *dequeue* is invoked on an empty queue); and the state of the object can be a sequence of elements that can be stored in single memory words of the computing system, a set we shall name Φ. In view of these properties, the *queue* type would be defined as the tuple $Q := (\Sigma, \mathcal{P}, Op, Op(p), Arg(\alpha), Res(\alpha), \delta))$ where:

- $\Sigma := \bigcup_{n \in \mathbb{N}} \Phi^n$, the sequences (possibly empty) of the elements of Φ.
- $Op := \{enqueue, dequeue\}$ and for any $p \in Q.\mathcal{P}$, $Op(p) := \{enqueue, dequeue\}$.
- $Arg(enqueue) := \Phi$ and $Arg(dequeue) := \emptyset$.
- $Res(enqueue) := \emptyset$ and $Res(dequeue) := \Phi$.
- $\delta(\sigma, p, enqueue, v) := (\sigma \circ v, \texttt{ack})$, and if σ is not the empty sequence, $\delta(\sigma, p, dequeue, \perp) := (tail(\sigma), head(\sigma))$, for $\sigma \in Q.\Sigma$, $p \in Q.\mathcal{P}$ and

$v \in Q.Arg(enqueue)$, noting that $\circ$ signifies concatenation, and that *head* outputs the first element, and *tail* outputs all elements but the first of a sequence.

Having defined the building blocks of our computational model, we can now talk about *algorithms*, *implementations*, and eventually provide a rigorous definition for a *linearizable* implementation.

Definition 2.3 (Algorithms)**.** In our computational model, an *algorithm* A is a specification of the following components:

- $\mathcal{P}$, namely the set of processes for which A specifies programs; that is, a sequence of instructions.
- $\mathcal{O}$, namely the set of objects used in the algorithm. Naturally, every object O in $\mathcal{O}$ has a type. The objects in $\mathcal{O}$ are all called *base objects* of the algorithm.
- Π_p, which for each $p \in \mathcal{P}$ is the program assigned to p by A.
- $Init_p$ defined for all $p \in \mathcal{P}$, namely the set of all possible initial states of a process p. This must be a non-empty subset of all possible states of p.
- $Init_O$ defined for all base objects $O \in \mathcal{O}$, namely the set of all possible initial states of an object O. This must also be a non-empty subset of all possible states of O; i.e., $O.\Sigma$ determined by the type of O.

In addition to the specification of these components, an algorithm must also satisfy the integrity condition that a process $p \in \mathcal{P}$ can invoke an operation α with argument λ on an object $O \in \mathcal{O}$ in a line of its program Π_p only if O's type supports p's invocation of α with argument λ. Expressed formally, this means it must hold that $p \in O.\mathcal{P}$, $\alpha \in O.Op(p)$, and $\lambda \in O.Arg(\alpha)$.

Definition 2.4 (Implementations and atomic implementations)**.** Consider an object of type $T := (\Sigma, \mathcal{P}, Op, Op(p), Arg(\alpha), Res(\alpha), \delta))$. An *implementation* of an object of type T initialized to a state $\sigma \in \Sigma$ is an algorithm A where:

- The algorithm specifies a program Π_p for each process p allowed to access the implemented object, and also encompasses a number of base objects. The *state* of the object being implemented is a function of the values stored in its base objects, whose initial states are configured to correspond to σ.
- The program Π_p of any process p consists of a special *main* method, as well as methods $\alpha_p(\lambda_p)$ for each operation $\alpha \in Op(p)$ with $\lambda \in Arg(\alpha)$.
- The main method of p consists of a single call: the non-deterministic selection of an operation $\alpha \in Op(p)$ and a corresponding argument $\lambda_p \in Arg(\alpha)$, followed by an invocation of the method $\alpha_p(\lambda_p)$. Furthermore, at the end of the execution of the invoked method, the process returns to its main method.
- Recalling the restrictions imposed by the concurrent computing system, the individual lines of the code (sequence of operations) for any method α_p consists of any number of operations on the *local variables*, and at most one on a single *shared object*; or a *return* line which makes the process return to its main method, and contains a response. We note that canonically, return lines can only be the final lines of methods.

Then, an algorithm A is said to implement an object of type T if it meets these criteria. We note that this is, for the most part, a syntactic definition that currently lacks a correctness condition. This correctness condition will be *linearizability*. However, before we can define the concept of linearizability, we must first establish a few more preliminary definitions, including that of an atomic implementation.

An *atomic implementation* is a trivial implementation that implements an object of type T, using an atomic object of the same type. More formally, an atomic implementation of an object of type T, is an implementation where a single T typed

base object O is sufficient, and for every process p and operation $\alpha \in Op(p)$ with $\lambda_p \in Arg(\alpha)$, the method $\alpha_p(\lambda_p)$ can be trivially defined in two lines of code as follows:

method $\alpha_p(\lambda_p)$
1. $res \leftarrow O.\alpha(\lambda_p)$
2. **return** *res*

Figure 1: Pseudocode description of *atomic implementation*

Of course, such implementations are only possible if O is readily available for use as an atomic object on our system, which more often than not is not the case, hence the interest in defining and leveraging linearizability.

Algorithms run over intervals of time, changing the states of objects and processes with every executed instruction. To capture this notion, we will introduce the concepts of *runs* and *histories*, which will later enable us to define *linearizability* in rigorous terms.

Definition 2.5 (Runs and histories)**.** Considering an arbitrary algorithm A, let us first define some concepts which will motivate the definition of runs and histories.

- A *configuration* of A is any assignment of values to the states of the objects and processes of A.
- An *initial configuration* of A is a configuration of A where for any process p, its state is in $Init_p$, and for any object O, its state is in $Init_O$.
- An *event* is a pair (p, a) where p is an arbitrary process, and a is any line from the sequence of instructions constituting its program Π_p.
- A *step* is a triple $(c, (p, a), c')$ where c is an arbitrary configuration of A, (p, a) is an event where a is the line pointed to by pc_p in c, and c' is the configuration obtained by p's execution of line a of its program, from the configuration c.

These definitions provide us with the necessary tools to represent the idea of how algorithms change the configuration of the system over the course of the execution of the instructions assigned to its processes.

A *run* of A is a sequence of configurations c_i and events (p_i, a_i) of the form $\langle c_0, (p_1, a_1), c_1, (p_2, a_2), \ldots \rangle$ such that c_0 is an initial configuration of A, and for each $i \geq 1$, $(c_{i-1}, (p_i, a_i), c_i)$ is a step of A. This sequence can be finite or infinite; however, if it is the case that it is finite, it must terminate with a configuration.

A *history* of A is the subsequence that contains all the events in a run of A. Then, for a *run* of the form $\langle c_0, (p_1, a_1), c_1, (p_2, a_2), \ldots \rangle$, the corresponding history would be of the form $\langle (p_1, a_1), (p_2, a_2), \ldots \rangle$.

Definition 2.6 (I/O events and I/O behaviors)**.** Let A be an implementation of an object typed T. An *I/O event* of A is an event (p, a) of A if a is either the invocation of a method of the implemented object (i.e., $\alpha_p(\lambda_p)$ for a process p and $\alpha \in Op(p)$ with $\lambda_p \in Arg(\alpha)$), or alternatively a return statement (i.e., "**return** *res*" for $res \in Res(\alpha)$ for some $\alpha \in Op(p)$ for a process p).

Then, an *I/O behavior* of the implementation A is a subsequence of a history of A that contains all the *I/O* events of the given history.

Definition 2.7 (Linearizable implementation)**.** A *linearizable implementation* of an object of type T is any implementation A where every *I/O behavior* of A is an *I/O behavior* of an atomic implementation of an object of type T.

This definition quite closely corresponds to our intuitive understanding of linearizability: actions "appearing" to take place instantaneously is equivalent to the correspondence of the *I/O events* within the *I/O behavior* of an implementation to *I/O events* in its atomic counterpart.

3 Linearizability as an invariant

To state the theorem we have established as part of our previous work, which relates the concept of linearizable implementations to the invariants of an algorithm, we must first define the notion of *transformations*.

Definition 3.1 (Transformations)**.** Consider the implementation A an object O of type $(\Sigma, \mathcal{P}, Op, Op(p), Arg(\alpha), Res(\alpha), \delta)$, initialized to $\sigma \in \Sigma$. A *transformation* of A is an algorithm A' that copies the behavior of A in its entirety, and additionally keeps track of the possible states of the implemented object O as well as the variety of ways how the methods executed by the processes can be linearized. This is achieved via the use of an *auxiliary variable* that does not affect the behavior of the algorithm (i.e., one with no effect on the code flow of any process, nor on the computation of any values used in the original implementation).

We note that every possible way in which the methods invoked by processes may linearize, determines a value for the implemented object's state, as well as the set of responses that the processes invoking these methods receive. In A', this information is kept track of with the use of an auxiliary variable T, specifically a set of tuples. Each tuple in T represents a particular linearization of the methods so far invoked on the implemented object, and consists of two components: one named *state* which represents the implemented object's state, and the other named *res*, a function which maps processes to the responses received by the invoking processes yet to terminate (return). In any transformation A' of A, the program $A'.\Pi_p$ for a process p is nearly identical to $A.\Pi_p$, with the addition that it can possibly update T in any line, under certain rules that we will soon describe.

Let us now define A' formally. A transformation A' of the implementation A is an algorithm such that:

- $A'.\mathcal{P} = \mathcal{P}\ (= A.\mathcal{P})$ and $A'.Init_p = A.Init_p$ for all $p \in \mathcal{P}$.
- $A'.\mathcal{O} = A.\mathcal{O} \cup \{T\}$.

  T is a set of tuples t with two components: $t.state \in \Sigma$, and $t.res$ in the set of functions from $\mathcal{P}$ to $\bigcup_{\alpha \in Op} Res(\alpha) \cup \{\perp\}$ such that for $p \in \mathcal{P}$, $t.res(p) \in Res(\alpha) \cup \{\perp\}$ for an $\alpha \in Op(p)$.

  For any process p, $t.res(p)$ represents the response the process p will receive under the conjectured linearization represented by the tuple; and this value is $\perp$ if no such response value is assigned. This case arises when the operation is yet to be linearized, or no operation is invoked by p.
- $A'.Init_Q = A.Init_Q$ for all $Q \in A.\mathcal{O}$, and $A'.Init_T = \{t\}$ where t is the tuple with $t.state = \sigma$ and $t.res : p \mapsto \perp$ for any process p. This makes intuitive sense, since at the initial configuration, no method invocations have yet occurred, and the state of the object must correspond to the value it has been initialized to.
- For every process $p \in \mathcal{P}$, $A'.\Pi_p$ contains the sequence of operations specified in each line of $A.\Pi_p$ in the same order, in addition to a potential update of T at any line subject to two integrity conditions.

We will now define the two integrity conditions as rules *T1* and *T2*:

- *Rule T1*: When a process p executes an instruction $a \in \Pi_p$ that is not in its *main* method or not a return statement, a tuple t' is in T after the instruction, only if there is a tuple $t \in T$ before the execution, for which there exists a (potentially empty) sequence of k distinct processes $\langle p_1, \ldots, p_k \rangle$, each executing a non-*main* method α_{p_i} (let the corresponding operation also be polymorphically denoted by α_{p_i}) with the argument λ_{p_i}, such that $t.res(p_i) = \perp$ for all

$i \in [k]$, with the following additional property:

Let s_0 denote $t.state$, and let $\langle (s_1, r_1), \ldots, (s_k, r_k) \rangle$ be a sequence of pairs in $\Sigma \times \bigcup_{\alpha \in Op} Res(\alpha)$, such that for all $i \in [k]$:

$$(s_i, r_i) = \delta(s_{i-1}, p_i, \alpha_{p_i}, \lambda_{p_i}).$$

Then, we insist that t' must be such that $t'.state = s_k$, and also that for all $i \in [k]$, $t'.res(p_i) = r_i$ and for all other processes q, $t'.res(q) = t.res(q)$.

We note that this rule essentially permits updates to T that take tuples $t \in T$, and for any tuple, conjectures an order in which a subset of non-linearized processes (that are actively executing a method outside of their *main* method) are to be linearized, and allows for the existence of tuples t' that respect this order of linearizations, with respect to how its state changes from the initial state of $t.state$, and the responses the processes receive.

This rule adds a lot of tuples into T, where each one is a possible linearization candidate, noting that many of which will not take place in the conjectured order, thereby resulting in many "invalid" tuples. The second rule allows us to deal with the invalid tuples: in particular, by discarding them.

– *Rule T2*: When a process p executes an instruction $a \in \Pi_p$ that is a return statement, T is updated so that it only contains tuples t' such that there is a tuple $t \in T$ where $t.res(p)$ equals the returned value, with t' being defined as the tuple with $t'.state = t.state$, $t'.res(p) = \perp$, and for all processes q other than p, $t'.res(q) = t.res(q)$.

We also refer to this rule as the *filtering rule*, as it filters out all tuples t whose conjectured response value for p does not correspond to the actual returned value. This value is then emptied (i.e.; overwritten by $\perp$) as the process is idle after the execution of the return line.

We note that, if the object implementation is indeed linearizable, then there

should in principle remain at least one valid tuple representing whichever linearization will take place. Otherwise, the *filtering rule* can potentially empty the set T, which would imply that there are no valid ways to linearize the methods invoked on the implemented object. We will now state a theorem that builds on this observation.

Having defined transformations, we can now state our claimed theorem, which connects *linearizable implementations* to invariants of transformations of said implementations. We note that an *invariant* of an algorithm is a statement that holds true in every configuration of every run of the algorithm.

Theorem 3.1. An implementation A of a typed object is linearizable if and only if $T \neq \emptyset$ is an invariant of some transformation A' of A.

For a discussion of the proof of this theorem which was the focus of our previous work, see the Appendix. With this theorem as the basis for our technique, to prove any implementation linearizable, we can simply identify a transformation of the implementation and then prove that $T \neq \emptyset$ is an invariant of this new algorithm. This has proven to be a quite valuable reduction that has enabled us to produce machine-verifiable proofs of linearizability for a number of algorithms, the most significant of which we will examine in the next section.

4 Herlihy-Wing queue algorithm

4.1 The original algorithm

In the paper where they identify and define the notion of *linearizability*, Herlihy and Wing [HW90] also provide a linearizable queue algorithm. The proof the algorithm's linearizability takes up the final third of their paper and extends well into the appendix, and requires careful reading to comprehend and verify; and therefore appeared to us as an ideal candidate for the application of our technique.

Here is the pseudocode for the algorithm, in the style of our definition for *implementation* and recalling our definition of the *queue* type from the previous section.

Initial values
$L = 1$; $i_p, j_p, l_p \in \mathbb{N}^+$ for all $p \in \mathcal{P}$,
Q: empty array of infinite length
initialized to $\perp$ values.

method $main_p()$
0. **while** *true*
   $\alpha \in Op(p)$, $\lambda_p \in Arg(\alpha)$
   $\alpha_p(\lambda_p)$

method $enqueue_p(v_p := \lambda_p)$
1. $i_p \leftarrow F\&I(L)$
2. $Q[i_p] \leftarrow v_p$
3. **return** `ack`

method $dequeue_p()$
4. $l_p \leftarrow L$
5. **if** $l_p = 1$ **goto** 4
   else $j_p \leftarrow 1$
6. $x_p \leftarrow FAS(Q[j_p], \perp)$
   if $x_p = \perp$
      if $j_p = l_p - 1$ **goto** 4
      else
         $j_p \leftarrow j_p + 1$
         goto 6
7. **return** x_p

Figure 2: The original Herlihy-Wing queue algorithm

We first note that *F&I* (shorthand for *fetch-and-increment*) and *FAS* (shorthand for *fetch-and-store*) are atomic operations that are available on most, if not every, concurrent computing system. By definition, *fetch-and-increment*(x) increments the value of x by 1 and returns its initial value, and *fetch-and-store*(x, v) stores v in the variable x, and returns x's initial value.

We shall now briefly explain the algorithm. L is a natural number that points to the foremost unused index of Q. An enqueuing process p reads L and stores it in i_p, then increments L. i_p is the index of Q that p has claimed for use, to store its argument v_p. As such, p simply writes v_p in $Q[i_p]$ and returns. A dequeuing process p, on the other hand, starts by reading L into l_p; at which point indices 1 through $l_p - 1$ are determined to be the set of indices of Q that p will iterate through to find an element to dequeue (specifically, the first non-$\perp$ value it finds). This also means that if $l_p = 1$, there are no indices to consider, and therefore we restart the procedure and go back to line 4. p iterates through the determined set of indices (using the loop variable j_p) and reads each value $Q[j_p]$ into x_p. If any non-$\perp$ value is detected, x_p is immediately returned; or if it is the case that $j_p = l_p - 1$, as this indicates that we have exhausted all considered indices without finding a stored value, we start the procedure anew.

We note that this is not an optimal algorithm. One can immediately notice that once elements stored at a certain index in Q are dequeued, the index is never reclaimed, hence unnecessarily increasing both the consumed amount of space, and the runtime of future dequeues. Herlihy and Wing also take note of this fact, and explain that this algorithm has been stated for the purpose of demonstrating a *linearizable* queue implementation. As such, they argue that as the optimization of the algorithm does not add to the discussion of the concept they define, this is not of interest for the paper.

4.2 Transforming the algorithm

We propose the following transformation of the Herlihy-Wing queue algorithm. Considering a process p at line 1, for each tuple $t \in T$, we allow the linearization of any subset of unlinearized enqueues (i.e., either p or any process q at lines 2 or 3 with $t.res(q) = \bot$) to create new tuples t' (possibly many). Otherwise, at line 6, if x_p is not $\bot$ (after the read), we simply linearize p to create a new tuple t' for each tuple $t \in T$, or leave T unchanged if this is not the case. We also leave T unchanged at every other line.

Let us first define a few expressions that will allow us to formally describe this transformation. In the following expressions, p and q are processes (and therefore are quantified over the set of processes $\mathcal{P}$), and t and t' are tuples as described in the definition of transformations (hence quantified over the domain of tuples respecting this definition).

$$\begin{aligned} \mathit{UnlinearizedEnqs}_p(t) &:= \{p\} \cup \{q : pc_q \in \{2,3\} \wedge t.res(q) = \bot\}, \\ \mathit{Filter}_p(r) &:= \{t' : \exists t \in T : t'.state = t.state \wedge t.res(p) = r \wedge t'.res(p) = \bot \\ &\qquad \wedge \forall q : q \neq p \implies t'.res(q) = t.res(q)\}, \\ \textsc{Perm}(S) &:= \{\sigma : \sigma \text{ is a permutation of the elements of } S\}. \end{aligned}$$

We note that *Filter* is a succinct shorthand that allows us to perform the action described in the *filtering rule*. Let us now formally describe this transformation in pseudocode. The initial values for all variables that appear in the original algorithm in **Figure 2** are unchanged, and the initial value of T is known to be $\{t\}$ where t is the tuple with $t.state = \langle\ \rangle$ (i.e. the empty sequence) and $t.res$ is the function $p \mapsto \bot$ for any process p, as per the definition of transformations.

method *main*$_p$()

0. **while** *true*
   $\alpha \in Op(p)$, $\lambda_p \in Arg(\alpha)$
   $\alpha_p(\lambda_p)$

method *enqueue*$_p$($v_p := \lambda_p$)

1. $i_p \leftarrow F\&I(L)$
   $T \leftarrow \{t' : \exists t \in T : \exists S \subseteq UnlinearizedEnqs_p(t) : \exists \sigma \in \text{PERM}(S) :$
   $t'.state = t.state \circ \langle v_{\sigma_1}, v_{\sigma_2}, \ldots, v_{\sigma_{|S|}} \rangle$
   $\wedge\, \forall q : (q \in S \implies t'.res(q) = \texttt{ack})$
   $\wedge\, (q \notin S \implies t'.res(q) = t.res(q))\}$
2. $Q[i_p] \leftarrow v_p$
3. **return** `ack`
   $T \leftarrow Filter_p(\texttt{ack})$

method *dequeue*$_p$()

4. $l_p \leftarrow L$
5. **if** $l_p = 1$ **goto** 4
   else $j_p \leftarrow 1$
6. $x_p \leftarrow FAS(Q[j_p], \bot)$
   if $x_p = \bot$
   if $j_p = l_p - 1$ **goto** 4
   else
   $j_p \leftarrow j_p + 1$
   goto 6
   else
   $T \leftarrow \{t' : \exists t \in T : t'.state = tail(t.state) \wedge t'.res(p) = head(t.state)$
   $\wedge\, \forall q : (q \neq p \implies t'.res(q) = t.res(q))\}$
7. **return** x_p
   $T \leftarrow Filter_p(x_p)$

Figure 3: A *transformation* of the Herlihy-Wing queue algorithm

We can easily confirm that this is a transformation of the Herlihy-Wing queue. All lines of the original algorithm in **Figure 2** appear unchanged in the new algorithm (and the explicit addition of the *else* case in line 6 does not change the flow of the algorithm, as there is originally an unstated skip), and the *filtering rule* is clearly respected at lines 3 and 7. At line 1, *Rule T1* is applied for every possible sequence of unlinearized enqueuing processes; and at line 6, *Rule T1* is applied with p as the only linearized process. By the definition of the *queue* object type, we can observe that the updating of the states of the tuples in T, as well as the responses assigned to processes in these tuples, correspond to how they are defined under *Rule T1*: the sequence of enqueuing processes append their arguments in the order which they are picked to be linearized and get `ack` responses, and a dequeuing process receives the head of the queue as a response, and sets the state to the tail of the queue. At all other lines, *Rule T1* is applied, with an empty sequence of processes.

Recalling **Theorem 3.1**, to prove that the Herlihy-Wing queue algorithm is a linearizable queue implementation, we want to show that $T \neq \emptyset$ is an invariant of this algorithm. To do so, we will establish an inductive invariant for the algorithm which implies this statement. We will finally machine-verifiably prove that the claimed invariant is indeed an inductive invariant, and one that implies $T \neq \emptyset$, using the TLA$^+$ Proof System.

We emphasize that if one merely wants to confirm that the algorithm is indeed linearizable, our machine-verified proof reduces this task to the verification of the fact that the proposed transformation in **Figure 3** is indeed a transformation of the algorithm (which we were able to assert within a single paragraph). Therefore, the invariant and its proof of correctness are, in fact, immaterial, so long as the invariance of the statement $T \neq \emptyset$ for the transformation is known to be machine-verifiably confirmed.

4.3 The invariant

We note that an *inductive invariant* is defined as an invariant that can be proven inductively. This proof by induction should consist of a proof that the invariant holds true for the initial configuration, and assuming the invariant holds true for an arbitrary configuration of the algorithm, that it holds true for the subsequent configuration after the execution of any line of the algorithm. Most invariants are not sufficiently robust, by themselves, to allow for this type of proof. Therefore, it is often the case that inductive invariants are a conjunction of multiple invariants of the algorithm.

Proposition 4.1. The suggested transformation of the Herlihy-Wing queue algorithm has an inductive invariant that implies $T \neq \emptyset$.

If we can machine-verifiably prove this claim correct, then by **Theorem 3.1**, we will have machine-verifiably shown the Herlihy-Wing queue implementation to be linearizable. Let us now attempt to identify a sufficiently strong inductive invariant, by first making a few observations about the transformation we proposed in **Figure 3**:

- From our definition of the transformation, it is obvious that for any process p at lines 0, 1, 4, 5, or 6, it must be the case that for any tuple $t \in T$, $t.res(p) = \perp$, as enqueuing processes are linearized after their first line, and dequeuing processes are linearized only as they head into line 7.
- Noting how i_p is determined at line 1, and that L is only ever incremented, it is trivially the case that for any process p at lines 2 or 3, $1 \leq i_p < L$.
- As L is atomically incremented at line 1 as i_p is determined for a process p (through the use of the *fetch-and-increment* operation), it is necessarily the case

that any two distinct processes p and q at lines 2 or 3 have different values in i_p and i_q.

- Noting how l_p is determined at line 4, it is also trivially the case that for any process p at line 5 and 6, $1 \leq l_p \leq L$.
- Furthermore, as j_p is a loop variable that iterates through indices 1 up to $l_p - 1$ as discussed in the previous section, we can extend this inequality to $1 \leq j_p < l_p \leq L$ for processes p at line 6.
- We also note that L denotes the foremost unused index of Q. Therefore, it must be the case that for any index k of Q greater than $L - 1$, $Q[k] = \bot$. Then, we can also observe that at line 2, $Q[i_p]$ must be empty, as L at the previous line was an unused index of Q.

In light of these observations, we can define the following statements, which we claim are invariants of the provided transformation.

$$
\begin{aligned}
Inv_0 &:= \forall p \in \mathcal{P} : pc_p = 0 \implies (\forall t \in T : t.res(p) = \bot) \\
Inv_1 &:= \forall p \in \mathcal{P} : pc_p = 1 \implies (\forall t \in T : t.res(p) = \bot) \\
Inv_2 &:= \forall p \in \mathcal{P} : pc_p = 2 \implies (1 \leq i_p < L) \wedge (Q[i_p] = \bot) \\
&\qquad \wedge (\forall q \in \mathcal{P} : (pc_q \in \{2,3\} \wedge q \neq p) \implies i_q \neq i_p) \\
Inv_3 &:= \forall p \in \mathcal{P} : pc_p = 3 \implies (1 \leq i_p < L) \\
&\qquad \wedge (\forall q \in \mathcal{P} : (pc_q \in \{2,3\} \wedge q \neq p) \implies i_q \neq i_p) \\
Inv_4 &:= \forall p \in \mathcal{P} : pc_p = 4 \implies (\forall t \in T : t.res(p) = \bot) \\
Inv_5 &:= \forall p \in \mathcal{P} : pc_p = 5 \implies (\forall t \in T : t.res(p) = \bot) \wedge (1 \leq l_p \leq L) \\
Inv_6 &:= \forall p \in \mathcal{P} : pc_p = 6 \implies (\forall t \in T : t.res(p) = \bot) \wedge (1 \leq j_p < l_p \leq L) \\
Inv_{S1} &:= \forall k \in \mathbb{N}^+ : k > L - 1 \implies Q[k] = \bot.
\end{aligned}
$$

Figure 4: Some invariants for the proposed Herlihy-Wing queue *transformation*

We claim that the conjunction of these invariants is indeed an inductive invariant of the transformation, yet not one that is robust enough to imply the result

we are interested in: that $T \neq \emptyset$. Most importantly, the invariant does not state anything about the existence of tuples in T; all quantifications over tuples t are universal, hence it is not necessarily the case that T is not empty. We therefore need to strengthen our invariant.

We will first introduce a few predicates that will provide us with tools to describe a significant fact about the algorithm, which will then sufficiently strengthen our invariant.

Consider the first line of the proposed transformation in **Figure 3**, and let us define a predicate, *GoodEnqSet*, that, given a set A of indices between 1 and $L-1$, determines whether the set of indices can correspond to a set of linearized *enqueue* operations (let us refer to such indices as linearized enqueue indices). First, if an index $k \in [L-1]$ corresponds to a non-empty element stored in Q (i.e., $Q[k] \neq \perp$), then it is linearized and must appear in A. Otherwise, an index k that appears in A and corresponds to an empty element in Q (i.e., $Q[k] = \perp$), must appear in A by the existence of a process q at line 2 that has claimed k as the index of Q it will write into. Let us formally define *GoodEnqSet*:

$$\begin{aligned} \mathit{GoodEnqSet}(A) := \forall k \in [L-1] : (Q[k] \neq \perp \implies k \in A) \wedge \\ ((Q[k] = \perp \wedge k \in A) \implies (\exists q \in \mathcal{P} : pc_q = 2 \wedge i_q = k)). \end{aligned}$$

Then, let us consider the responses processes should receive under specific linearizations. If a process q is at line 3 or 7, it must have linearized and must have been assigned a response value of either `ack` or x_q respectively. Moreover, it must have been assigned a response value of `ack` at line 2 if q, or some other process, linearized q's *enqueue* operation as part of their update to T in line 1. With this observation, let A be a conjectured set of linearized enqueue indices of Q between 1

and $L-1$. Then, any process q at line 2, whose selected index i_q appears in A, must also be linearized and be assigned a response value of `ack`. No other processes are assigned a response value. In view of these observations, for a conjectured set of linearized enqueue indices A, and a tuple $t \in T$, let us define the predicate $\mathit{GoodRes}(A, t)$, which indicates that the *res* function of t has response values that correspond to the linearizations conjectured by A:

$$\mathit{GoodRes}(A,t) := \forall q \in \mathcal{P} : t.\mathit{res}(q) = \begin{cases} \texttt{ack} & \text{if } pc_q = 2 \ \wedge \ i_q \in A \\ \texttt{ack} & \text{if } pc_q = 3 \\ x_q & \text{if } pc_q = 7 \\ \bot & \text{otherwise.} \end{cases}$$

Similarly to how we define a valid correspondence between responses in tuples and responses returned as a result of particular conjectured linearizations in *GoodRes*, we can also define the correspondence between states of tuples, and states of the implemented object obtained as a result of said linearizations. If σ is a conjectured order in which linearized enqueue indices of Q are linearized (i.e., a permutation of the set A discussed in the preceding paragraph), then the corresponding state of the implemented object should be a sequence of values from Q pointed to by the indices; or if an index points at an empty component of Q, the argument of the process whose index appears in the sequence; in the order the indices appear in σ. In light of these observations, for a conjectured sequence of linearized enqueue indices σ (in the order in which they are conjectured to linearize), and a tuple $t \in T$, let us also define the predicate $\mathit{ValuesMatchInds}(\sigma, t)$, which indicates that the *state* of t corresponds to the order of linearizations conjectured by σ:

$$\mathit{ValuesMatchInds}(\sigma,t) :=$$
$$t.\mathit{state} = \left(k \in \{1,\ldots,\mathit{length}(\sigma)\} \mapsto \begin{cases} Q[\sigma_k] & \text{if } Q[\sigma_k] \neq \bot \\ v_{\epsilon q \in \mathcal{P} \,:\, pc_q = 2 \,\wedge\, i_q = \sigma_k} & \text{otherwise} \end{cases} \right),$$

where the statement "$\epsilon q \in \mathcal{P} : pc_q = 2 \wedge i_q = \sigma_k$" should be read as a process q that satisfies $pc_q = 2 \wedge i_q = \sigma_k$, if it exists. This operator is occasionally known as Hilbert's ϵ-operator [AZ20]. We note that such processes will exist in the case of correctly conjectured linearizations; i.e., if t is a tuple which corresponds to a sequence of linearizations σ that will in fact take place, then there will be a process q that can be picked for this comparison.

For the last definition needed to establish the final invariant to complete our inductive invariant, let us once again consider a conjectured set A of linearized enqueue indices of Q, and σ a conjectured permutation of A that corresponds to the order in which they are linearized. Let us reflect about what makes a σ a potentially valid linearization. If σ is simply the non-empty indices of Q in the order they appear in Q, it should clearly represent a valid linearization; an incoming dequeuing process can execute a number of dequeues and indeed empty the queue in this order, meaning that this sequence corresponds to a potentially valid state of the implemented object (the state specifically being the elements of Q stored at the given sequence of indices, in the order of the sequence). Let σ instead not follow this order, and assume it contains two indices m, n such that $n < m$ yet $\sigma_m < \sigma_n$ (meaning that m is an index lower than n but appears later than n in the sequence σ). In this case, if σ_m points to an empty component in Q (i.e. $Q[\sigma_m] = \perp$), this is once again a potentially valid ordering of the indices, as a dequeuing process might very well pass through σ_m as it loops through indices before reaching σ_n, since nothing is stored at $Q[\sigma_m]$. If it is the case that $Q[\sigma_m] \neq \perp$, however, this is not immediately a potentially valid ordering of indices. To claim that this is indeed a possible ordering, we must assert that there exists a dequeueing process p, which is past the index σ_m (i.e. $\sigma_m < j_p$) in its loop, such that $\sigma_n < l_p$ (which would otherwise fall outside the scope of indices p could dequeue from).

To capture this idea, let an *inversion* for a sequence of indices σ denote a pair of indices m, n for which $n < m, \sigma_m < \sigma_n, Q[\sigma_m] \neq \perp$. For a sequence to be a potentially valid sequence of linearized enqueues, the inversions it contains must all be *justified*. In light of our discussion in the previous paragraph, a *justified inversion* is an inversion for which there is a dequeuing process p at line 6, for which $\sigma_n < l_p$ and $\sigma_m < j_p$. Then, a sequence of conjectured enqueuing indices σ for which every pair of indices is either not an inversion, or is a justified inversion, is a valid linearization of enqueues. Let us define this notion as a predicate, named *JInvSeq*:

$$\begin{aligned} \mathit{JInvSeq}(\sigma) := \forall m, n \in \{1, \ldots, \mathit{length}(\sigma)\} : (n < m \wedge \sigma_m < \sigma_n \wedge Q[\sigma_m] \neq \perp) \\ \implies (\exists p \in \mathcal{P} : pc_p = 6 \wedge \sigma_n < l_p \wedge \sigma_m < j_p). \end{aligned}$$

As every sequence σ of conjectured enqueuing indices which satisfies this property is a valid potential linearization, it must be witnessed in T with a corresponding tuple. We indeed have the suitable tools to describe what a corresponding tuple would be, thanks to the predicates *GoodRes* and *ValuesMatchInds*. Let us then formally describe our final invariant:

$$\begin{aligned} \mathit{Inv}_{S2} := \forall A \subseteq \{1, \ldots, L-1\} : \mathit{GoodEnqSet}(A) \\ \implies (\forall \sigma \in \textsc{Perm}(A) : (\mathit{JInvSeq}(\sigma) \\ \implies (\exists t \in T : \mathit{GoodRes}(A, t)) \wedge \mathit{ValuesMatchInds}(\sigma, t))). \end{aligned}$$

We then claim that the following is an *inductive invariant* of the proposed transformation of the Herlihy-Wing queue algorithm:

$$\mathit{Inv} := \mathit{Inv}_1 \wedge \mathit{Inv}_2 \wedge \ldots \wedge \mathit{Inv}_6 \wedge \mathit{Inv}_{S1} \wedge \mathit{Inv}_{S2}.$$

Figure 5: The inductive invariant for the Herlihy-Wing queue *transformation*

We note that this invariant also implies the statement $T \neq \emptyset$, hence proving **Proposition 4.1**. We can simply show this with the help of the conjunct Inv_{S2}. Consider the set A of all non-empty indices in Q in the order σ they appear in Q. This is clearly a valid set of conjectured enqueues by the definition of *GoodEnqSet*; and contains no inversions, hence vacuously fulfilling $JInvSeq(\sigma)$. This is sufficient for Inv_{S2} to imply the existence of a tuple $t \in T$; and the tuple t is, in addition, a tuple that represents a possible linearization by the definitions of *GoodRes* and *ValuesMatchInds*.

As **Proposition 4.1** is proven by the invariant *Inv*, to complete the machine-verifiable proof that the Herlihy-Wing queue algorithm is linearizable, all that remains is to machine-verifiably prove that *Inv* is an inductive invariant of the proposed transformation in **Figure 3**.

In the next section, we will discuss the proof of this inductive invariant as well as the proposition that it implies $T \neq \emptyset$, which we have been able to write and confirm in the TLA$^+$ Proof System [Cha+08]. We once again underline that in light of our machine-verified proof, and **Theorem 3.1**, a reader who wants to assure themselves of the linearizability of the implementation, only needs to verify that the transformation we proposed is indeed a transformation of the Herlihy-Wing queue algorithm, which we were able to establish within a single paragraph in the previous section. Therefore, the amount of effort needed to motivate and also prove the invariant, clearly exhibits the usefulness of our technique by illustrating the sheer amount of responsibility it takes away from the reader with regard to the verification of the claim.

5 The machine-verified proof of correctness

5.1 Specifying the algorithm in TLA$^+$

TLA$^+$ is a specification language, designed by Leslie Lamport, for formally describing and reasoning about distributed algorithms [Lam02; Mer08]. In TLA$^+$, algorithms are specifically described as formulas in the *temporal logic of actions* (also developed by Lamport), a variant of *linear temporal logic* [GR22], which itself is an extension of propositional logic, with the addition of rules and symbols for describing and reasoning about propositions expressed in terms of time.

TLA$^+$ allows us to describe algorithms in terms of predicates on *states*, and predicates on pairs of states, which correspond to *actions*. *States* are logical propositions that describe the configuration of the system, by stating the current value of each variable of the system. *Actions* can be understood as predicates on pairs of states: each line in the code of an algorithm can be thought of as a distinct *action*, which is a logical proposition that describes the state after the execution of this instruction, conditioned on the state before this execution. The values of variables before and after the action are distinguished by the use of *primed* and *unprimed* variables. In *action* formulas, the unprimed variable v refers to the value of v in the current configuration, whereas the primed variable v' is its value in the next configuration, obtained by the execution of the instruction that corresponds to the specific action.

By convention, algorithms are represented by the following logical expression named conventionally named *Spec* (shorthand for *specification*) in TLA$^+$:

$$Spec := Init \wedge \Box[Next]_{vars}.$$

Init, referred to as the *initial-state predicate*, as its name suggests, is a proposition that describes the possible set of values for each variable of the algorithm in an initial configuration. For instance, for the transformation proposed in **Figure 3**, *Init* is a proposition which assigns values to the shared variables L, Q, the shared auxiliary variable T and the local variables $pc_p, i_p, j_p, l_p, x_p, v_p$ for every process p. L, Q, T and pc_p have precise initial values as previously discussed, so their assignment entails an equality, whereas the other variables are arbitrarily initialized, and hence the assignment only entails a set membership (in other words, the proposition that they belong in their domains; e.g. $i_p \in \mathbb{N}^+$). *Init* is hence a logical proposition that holds true in every initial configuration of the algorithm.

Next, referred to as the *next-state action*, is a proposition that describes every possible change to the configuration of the system as the result of the execution of an instruction, from the set of instructions that compose the algorithm. In our concurrent computing system, such a change occurs when a process $p \in \mathcal{P}$ executes a line $a \in \Pi_p$. Then, for the proposed transformation, assuming we have defined action predicates that describe the execution of lines 0 through 7 for a process p, $L00(p), L01(p), \ldots, L07(p)$, we can describe *Next* with the following formula:

$$Next := \exists p \in \mathcal{P} : L00(p) \vee L01(p) \vee \ldots \vee L07(p).$$

Each line can be described as an action formula involving a number of conjuncts, each of which specify the *primed* variables for every variable of the algorithm (i.e.,

the values of each variable in the new configuration obtained by the execution of the line). The transitions between lines are simply changes to the program counter, and therefore are captured by the specification of pc'_p. *If-else* constructions, or other constructs that describe the program's flow, can also be described in the language of TLA$^+$. It is important to note that a process should be able to execute a line, only if its program counter points at that line. Therefore, it is imperative that these line actions hold true only if pc_p corresponds to the number of the line: for instance, the action $L01(p)$ should include $pc_p = 1$ as one of its many conjuncts.

vars is a shorthand for tuple containing all variables of the algorithm, and $\Box$ is an operator in temporal logic which means "always". $[Next]_{vars}$ stands for $Next \vee vars' = vars$, which describes every possible action, and possibly no actions happening. Then, we can see that $Spec = Init \wedge \Box[Next]_{vars}$ describes an algorithm for which *Init* holds true in the initial configuration, and for every run, *Next* holds true for each step.

The TLA$^+$ specification for the transformation proposed in **Figure 3** can be found in the Appendix. Having described what the specification of this algorithm in TLA$^+$ would entail, and having provided this specification, we shall now discuss the machine-verified TLAPS proof that *Inv* is an invariant of the transformation, strong enough to fulfill the requirements of **Theorem 3.1**, thereby machine-verifiably proving the original algorithm linearizable.

5.2 The TLAPS proof

In TLA$^+$ notation, the fact that *Inv* is an *invariant* of the algorithm specified by *Spec* would be denoted by the formula $Spec \implies \Box Inv$, meaning that in every run of the algorithm specified by *Spec*, *Inv* always holds true. Given how algorithms are specified in TLA$^+$, the natural proof technique to prove this claim would be to prove it by induction, as the *initial-state predicate* corresponds to a base case, and a *next-state action* can be conceived as the induction step.

TLAPS is a proof system for the verification of TLA$^+$ proofs. A TLA$^+$ proof is a collection of claims expressed in TLA$^+$ organized hierarchically, with each claim containing a statement that is either unsupported or supported by a collection of referenced facts. TLAPS verifies that the supplied hierarchy of claims proves the theorem if the claims are true, and subsequently verifies that the statement of each justified claim is in fact implied by the listed facts [Mic15]. If a TLA$^+$ theorem's proof contains no invalid assertions, then TLAPS certifies the validity of the theorem by examining the proof, through the use of three backend automated theorem provers: *Z3* [MB08], *Zenon* [BDD07], and *Isabelle* [PN94].

Then, a TLAPS proof of the claim that *Inv* is an invariant of the algorithm, can be thought of as a proof by induction, where the steps of the outermost level of the hierarchy of claims consist of the claim that *Inv* holds true in the initial configuration; and the claim that assuming *Inv* holds true at an arbitrary configuration, *Inv*' (*Inv* in the next configuration obtained by executing any action in *Next*) also holds true. This structure is precisely why we needed to ensure that *Inv* is an inductive invariant. We also note that the second step would be proven by the proof of the claims for each possible action in *Next* individually (one assuming *Inv* and *L00*,

one assuming *Inv* and *L01*, etc.). Each such claim is a step in the second level of the hierarchy, under the second step of the first level.

To prove this invariant correct in TLAPS, in TLA$^+$ notation, we must prove that $Init \implies Inv$ and $Inv \wedge [Next]_{vars} \implies Inv'$. We note that in TLA$^+$, variables are not typed; and although *Init* either explicitly defines variables or provides their domains, in the second step, $Inv \wedge [Next]_{vars}$ does not necessarily provide any information about the types of the variables, which is something that needs to be known for the claims to hold true. Therefore, when working on inductive invariants in TLAPS, it is often the case that we need to introduce another conjunct to the invariant, which asserts the *type correctness* of the variables. This invariant, conventionally named *TypeOK*, states the domain of every variable of the algorithm. Let us redefine *Inv*, as *Inv* strengthened by this type correctness invariant.

We have now established the structure of the TLAPS proof for this invariant. To produce a full TLAPS proof of correctness, what remains to be done is to provide the proofs for these claims, in the hierarchical format supported by TLAPS.

We have indeed produced a proof of the theorem $Spec \implies \Box\, Inv$ that has been certified by TLAPS. In addition, we produced a proof of the claim that $Inv \implies T \neq \emptyset$ and therefore $Spec \implies \Box\, T \neq \emptyset$, within the TLAPS framework. For the proof of correctness for the inductive invariant *Inv* of the proposed transformation, the proofs of the conjuncts *TypeOK*, Inv_0 through Inv_6, and Inv_{S1} are fairly straightforward, as T is unchanged in most steps, and it is relatively simple to reason about the behaviors of the other variables. The proof of the invariance of Inv_{S2}, although quite elaborate, boils down to proofs by construction for each line, argued via careful witness picking.

This proof is publicly available on a GitHub repository [YV22], and consists of nearly 3500 lines of TLA$^+$ code. For comparison, the specification of the algorithm, available in the Appendix, had only required 140 lines. The proof of the claim that *Inv* remains fulfilled after the execution of Line 1 by a process is the longest component of the proof, taking over 1200 lines, in large part due to the elaborate definition of T at that line. We note that the proof of the claim that $Inv \implies T \neq \emptyset$, which was briefly discussed, requires the *well-ordering theorem*, a TLAPS proof of which is also provided alongside the two other proofs.

The sheer size of the proof makes it difficult for TLAPS to certify the proof in a single run: on a mid-range 2021 laptop and using 1 GB of RAM, it takes TLAPS around 50 minutes to examine the entire proof for the first time, which results in some uncertified steps in the lower steps of the hierarchy that need to be run individually to be certified. This second run requires another 30 minutes on the same machine.

By **Theorem 3.1**, this proof therefore machine-verifiably proves the linearizability of the original algorithm.

6 Conclusion and future work

To summarize, we have formally presented a technique which allows for the reduction of the task of proving linearizable object implementations correct, to the proof of an inductive invariant for a *transformation* of the proposed implementation, one that specifically implies $T \neq \emptyset$. As proof systems, such as the TLA+ Proof System we chose to work with, are apt at handling proofs of correctness for statements of this type, we conjectured that in principle, this reduction allows for the production of machine-verified proofs of linearizability. This, in turn, considerably reduces the amount of responsibility on the shoulders of a human verifier, who only wants to assure themselves of the linearizability of the implementation.

We then applied this technique, which thus far had only been applied to smaller-scale problems, to a produce a machine-verified proof of linearizability (in TLAPS) for a significant linearizable algorithm whose proof of correctness is known to be challenging: the Herlihy-Wing queue algorithm. We described the transformation and the invariant that allowed for the application of our technique in detail to provide insight, however noting that the invariant itself is not of interest for the human verifier by virtue of our technique. The applicability of the technique to such an algorithm is a strong indicator of the generalizability of the technique, to prove many more linearizable algorithms correct, from the literature of the domain of concurrent algorithms.

As future work, with the insight we have gained on TLA$^+$ and TLAPS as we

worked on this problem, we plan to improve the TLAPS proofs we have written so far as part of our research project; by optimizing the interactions between TLAPS and its backend provers by invoking them more intelligently in places where the proof might be sped up, factorizing the proof further to shorten its length and the number of claims to be proven, and making more frequent use of advanced constructs (such as HIDE, which allows for the ”hiding” of the definitions of complicated logical expressions, thereby accelerating the verification of the proof in places where these definitions are immaterial) available in TLAPS.

Furthermore, we intend to apply this technique to more algorithms in the future, and create a library of proofs within our framework, for a number of linearizable algorithms (such as Peterson’s algorithm for the *readers-writers problem* [Pet83]).

Appendix

A.1 Proof of Theorem 3.1

Let us recall the statement of the theorem:

Theorem 3.1. An implementation A of a typed object is linearizable if and only if $T \neq \emptyset$ is an invariant of some transformation A' of A.

We note that within all the claims made referring to this theorem, as we only used the invariance of $T \neq \emptyset$ for some A' to infer the linearizability of A, only one of the two directions of the theorem is necessary for the purposes of this paper. Let us state this direction of the theorem explicitly:

Theorem A.1. An implementation A of a typed object is linearizable if $T \neq \emptyset$ is an invariant of some transformation A' of A.

For the reason stated above, we will only address the proof of **Theorem 3.1** in this direction in detail. However, in broad strokes, to obtain a transformation A' from an implementation A, one would identify the linearization points of the algorithm A, and have the transformation A' explicitly linearize each operation at the line corresponding to their linearization point. $T \neq \emptyset$ would then have to be an invariant of this algorithm, as $T = \emptyset$ holding true at some configuration would indicate that there are no valid ways to linearize the history of operations that brought the implemented object to this specific configuration.

Let us now address the proof of **Theorem A.1**. This proof was first presented in my unpublished undergraduate thesis, and adjusting for changes in notation and a few clarifications, the proof we will discuss now is largely identical. We will, in large part, be helped by a lemma we will state shortly.

Let σ denote the initial state of the object implemented by both A and A', and let D denote the type of this object. Additionally, note that in any configuration c in any run of an implementation A (or its transformation A'), we can identify the method being executed by any process p by looking into $c.pc_p$ and the program $A.\Pi_p$, and we can similarly identify the argument the method is invoked with. Let us denote the method being executed by p at the configuration c as $c.\mu_p$ (either α_p for some operation α, or $main_p$), and let us denote the argument as $c.\lambda_p$. Finally, for any method $c.\mu_p$, let $(c.\mu_p).k$ denote its k^{th} line. We can now state the lemma.

Lemma A.2. If $T \neq \emptyset$ is an invariant of A', then for each run R of A' that ends in a configuration c, for any $t \in c.T$, there exists a run R_a of the atomic implementation of D initialized to σ, that ends in a configuration c_a and satisfies the following conditions for any process $p \in \mathcal{P}$:

- The *I/O behaviors* of R and R_a are identical.
- $t.state = c_a.O$ (i.e. the object's state), where O is the base object in the atomic implementation.
- If $t.res(p) = \perp$ and $c.\mu_p = main_p$, then $c.pc_p = c_a.pc_p = main_p.1$
- If $t.res(p) = \perp$ and $c.\mu_p \neq main_p$, then $c_a.pc_p = (c.\mu_p).1$.
- If $t.res(p) \neq \perp$, then $c_a.pc_p = (c.\mu_p).2$.

We underline that the methods here are the methods in the atomic implementation. Let us now prove this lemma, which we claim can be proven by induction on the number of steps in a run R of A'. For the base case, consider a run $R = \langle c \rangle$,

consisting of only an initial configuration. Let R_a be a run of the atomic implementation which also only consists c_a. The *I/O behaviors* of R and R_a are identical, as they are empty sequences. Furthermore, $t.state = c_a.O = \sigma$ by initialization. For any process $p \in \mathcal{P}$, it is clearly the case that $t.res(p) = \perp$, and as c is an initial configuration, $c.pc_p = main_p.1$, completing the proof of the claim for the base case.

For the induction step, let us assume that the claim holds for runs of A' with n steps. Consider a run $R' := \langle c_0, (p_1, a_1), c_1, \ldots, c_n, (p_{n+1}, a_{n+1}), c_{n+1} \rangle$ of A'. We know by our inductive hypothesis that for $\langle c_0, (p_1, a_1), \ldots, c_n \rangle$, there exists a run R_a of the atomic implementation which ends in a configuration c_a such that:

- The *I/O behaviors* of R and R_a are identical.
- $t.state = c_a.O$.
- For any $p \in \mathcal{P}$:

$$c_a.pc_p = \begin{cases} main_p.1 & \text{if } t.res(p) = \perp \text{ and } c_n.\mu_p = main_p \\ (c_n.\mu_p).1 & \text{if } t.res(p) = \perp \text{ and } c_n.\mu_p \neq main_p \\ (c_n.\mu_p).2 & \text{if } t.res(p) \neq \perp. \end{cases}$$

In order to prove the claim for R', we need to demonstrate the existence of a run R'_a of the atomic implementation with the desired properties. There are three cases to examine, depending on the line a_{n+1}. We note that a_{n+1} could be a method invocation from $main_{p+1}$, a return statement, or any other line of a method of p.

First, let us assume that a_{n+1} is a method invocation from $main_{p+1}$. At this point, neither *Rule T1* or *T2* applies, and it is also necessarily the case that $t.res(p_{n+1}) = \perp$. As neither rule applies, $c_{n+1}.T = c_n.T$. Let $R'_a. = R_a \circ \langle (p_{n+1}, a_{n+1}), c'_a \rangle)$. As (p_{n+1}, a_{n+1}) is an *I/O event* for A' and for the atomic implementation, the I/O behaviors of R' and R'_a are identical. By our hypothesis, we were also given that $t.state = c_a.O$. As the invocation does not affect the base object's state in R'_a, we note that $t.state = c_a.O = c'_a.O$. After the execution of line a_{n+1}, $c_{n+1}.pc_p = c'_a.pc_p =$

$(c_{n+1}.\mu_p).1$. Furthermore, the program counters of all other processes remain unchanged. The claim is then clearly correct for this case.

Secondly, let us assume that a_{n+1} is a return statement. By the *filtering rule*, we know that for $t \in c_{n+1}.T$, there is a tuple $u \in c_n.T$ where $u.res(p_{n+1}) \neq \perp$, $u.res(q) = t.res(q)$ for all other processes q, and $t.state = u.state$. Let $R'_a = R_a \circ \langle (p_{n+1}, a_{n+1}, c'_a \rangle$. As (p_{n+1}, a_{n+1}) is an *I/O event* for A' and for the atomic implementation, the I/O behaviors of R' and R'_a are identical. By our hypothesis, we were also given that $u.state = c_a.O$. As the return line does not affect the base object's state in R'_a, we note that $t.state = u.state = c_a.O = c'_a.O$. After the execution of line a_{n+1}, $c_{n+1}.pc_p = c'_a.pc_p = main_p.1$ and $t.res(p) = \perp$. Furthermore, the program counters of all other processes, as well their response values in all tuples remain unchanged. The claim is then also correct for this case.

Finally, let us assume that a_{n+1} is any other line. By *Rule T1*, we know that there are two possible cases for the tuple $t \in c_{n+1}.T$:

- $t \in c_n.T$, which would indicate that the sequence of processes linearized by the rule is the empty sequence. Then, let $R'_a = R_a$. As (p_{n+1}, a_{n+1}) is not an *I/O event* and $t.res(p_{n+1}) \neq \perp$, it should be evident that R'_a is a run with the desired properties.
- There is a tuple $u \in c_n.T$ for which there exists a sequence of processes $\langle q_1, q_2, \ldots, q_k \rangle$ with $u.res(q_i) = \perp$ for all $1 \leq i \leq k$, and a sequence of states $\langle s_1, s_2, \ldots s_k$, such that $t.state = s_k$ and:

$$i. \quad \delta(u.state, q_1, c_n.\mu_{q_1}, c_n.\lambda_{q_1}) = (s_1, u.res(q_1)),$$

$$ii. \quad i \in \{2, \ldots, k\} \implies \delta(s_{i-1}, q_i, c_n.\mu_{q_i}, c_n.\lambda_{q_i}) = (s_i, u.res(q_i)).$$

By the inductive hypothesis, we are given that for all processes q_i, $c_a.pc_{q_i} = (c_n.\mu_{q_i}).1$. Let $R'_a = R_a \circ \langle (q_1, (c_n.\mu_{q_1}).1), c'_1, \ldots, (q_k, (c_n.\mu_{q_k}).1), c'_k \rangle$. To maintain our notation

consistent, let $c'_0 = c_a$. We note that neither the step added to R to establish R', nor any of the steps added to R_a to establish R'_a are *I/O events*. Therefore, the I/O behaviors of R' and R'_a are also identical.

The invocation of $O.(c_n.\mu)(c_n.\lambda_{q_i})$ at the first line of $c_n.\mu_p$ from a configuration c'_{i-1} to reach a next configuration c'_i when (the state of) $O = s_{i-1}$, results in the object's state being the next state in the aforementioned sequence of states, i.e. $c'_i.O = s_i$. Then, as $u.state = s_0, t.state = s_k$, and $u.state = c'_0.O$ by our inductive hypothesis, it must also be the case that $t.state = c_a.O$.

In addition, following the execution of the sequence of events added to R_a to define R'_a, it is the case that $t.res(p) \neq \perp$. Furthermore, since for all processes q_i for $1 \leq i \leq k$, we have $c_a.pc_{q_i} = (c_n.\mu_{q_i}).1$, it must then be the case that $c'_k.pc_{q_i} = (c_n.\mu_{q_i}).2$. We also note that the program counters of all other processes, as well their response values in all tuples remain unchanged. This completes the proof of the claim for the third and the final case, hence completing the proof of **Lemma A.2**.

We claim that the following lemma, is a corollary to **Lemma A.2**:

Lemma A.3. If $T \neq \emptyset$ is an invariant of A', A' is a linearizable implementation of D.

Assuming $T \neq \emptyset$ is an invariant of A', we can infer from **Lemma A.2** that for each run R of A', there is a run R_a of the atomic implementation of D, such that the *I/O behaviors* of R and R_a are identical. As the I/O behavior of any run of A' is then also an I/O behavior of the atomic implementation, by the definition of linearizable implementations, we can conclude that A' is a linearizable implementation of the object type D.

Lemma A.4. If A' is a linearizable implementation of D, A is a linearizable implementation of D.

We note that for any run $R' = \langle c'_0, (p_1, a_1), \ldots, (p_n, a_n), c'_k \rangle$ of A', there is a run $R = \langle c_0, (p_1, a_1), \ldots, (p_n, a_n), c_k \rangle$ of A such that for all non-T variables v and all $1 \leq i \leq k$, it is the case that $c_i.v = c'_i.v$. Hence, the *I/O behaviors* of both runs are identical. Therefore, if A' is a linearizable implementation of D, so is A.

By **Lemmas A.3** and **A.4**, we have the proof of **Theorem A.1**.

A.2 The TLA$^+$ specification of the transformation

MODULE *HWQueue*

EXTENDS *Integers*, *Sequences*, *FiniteSets*

CONSTANTS ACK, BOT
VARIABLES pc, L, Q, i, j, l, x, v, T
$vars \triangleq \langle pc, L, Q, i, j, l, x, v, T\rangle$

Useful definitions
$PosInts \triangleq Nat \setminus \{0\}$
$ProcSet \triangleq Nat \setminus \{0\}$

Set of all permutations of items of S.
$Perm(S) \triangleq \{f \in [1 \,..\, Cardinality(S) \to S] : \forall\, w \in S : \exists\, q \in 1 \,..\, Cardinality(S) : f[q] = w\}$

Set of all sequences containing items from S.
Originally implemented in *Sequences*.
$Seqs(S) \triangleq \text{UNION } \{[1 \,..\, n \to S] : n \in Nat\}$

Defined Tail and Head
$Hd(s) \triangleq \text{IF } s \neq \langle\rangle \text{ THEN } s[1] \text{ ELSE } BOT$
$Tl(s) \triangleq [k \in 1 \,..\, (Len(s) - 1) \mapsto s[k+1]]$

Domain of all tuples: state is a sequence of naturals,
response is a natural, ACK, or BOT for each process.
$$\begin{aligned} TupleDomain \triangleq\ & [state : Seqs(Nat),\\ & \ res\ \ : [ProcSet \to Nat \cup \{ACK,\ BOT\}]] \end{aligned}$$

Set of unlinearized enqueues (*i.e.* no response at line 2, 3) given a linearization tuple t.
$UnlinearizedEnqs(t) \triangleq \{q \in ProcSet : pc[q] \in \{2, 3\} \land t.res[q] = BOT\}$

Set of linearization tuples after process p returns r.
(Filter non-r responses, and reset the response field).
$$\begin{aligned} Filter(p, r) \triangleq \{u \in TupleDomain : \exists\, t \in T :\ & \land\, t.res[p] = r\\ & \land\, u.state = t.state\\ & \land\, u.res[p] = BOT\\ & \land \forall\, q \in ProcSet : q \neq p \Rightarrow\ u.res[q] = t.res[q]\} \end{aligned}$$

Concatenation to sequence
$$\begin{aligned} Concat(s1, s2) \triangleq [k \in 1 \,..\, (Len(s1) + Len(s2)) \mapsto \text{IF } & k \leq Len(s1)\\ & \text{THEN } s1[k]\\ & \text{ELSE } s2[k - Len(s1)]] \end{aligned}$$

Algorithm and invariants

Define a (response-wise) well-formed tuple
$$\begin{aligned} tTypeOK(t) \triangleq\ \land \forall\, p \in ProcSet :\ & \land\, pc[p] \in \{0,\ 1,\ 4,\ 5,\ 6\} && \Rightarrow t.res[p]\ = BOT\\ & \land\, pc[p] \in \{2,\ 3\} && \Rightarrow t.res[p] \in \{ACK, BOT\}\\ & \land\, pc[p] = 7 && \Rightarrow t.res[p] \in Nat \cup \{BOT\} \end{aligned}$$

$$
\begin{array}{lll}
\mathit{Init} & \triangleq & \land\ pc = [p \in \mathit{ProcSet} \mapsto 0] \\
& & \land\ L\ \ = 1 \\
& & \land\ Q\ \ = [\mathit{idx} \in \mathit{PosInts} \mapsto \mathit{BOT}] \\
& & \land\ i\ \ \in [\mathit{ProcSet} \to \mathit{PosInts}] \\
& & \land\ j\ \ \in [\mathit{ProcSet} \to \mathit{PosInts}] \\
& & \land\ l\ \ \in [\mathit{ProcSet} \to \mathit{PosInts}] \\
& & \land\ x\ \ \in [\mathit{ProcSet} \to \mathit{PosInts} \cup \{\mathit{BOT}\}] \\
& & \land\ v\ \ \in [\mathit{ProcSet} \to \mathit{PosInts}] \\
& & \land\ T\ = \{[\mathit{state} \mapsto \langle\rangle, \\
& & \qquad\ \mathit{res}\ \ \mapsto [p \in \mathit{ProcSet} \mapsto \mathit{BOT}]]\}
\end{array}
$$

$$
\begin{array}{lll}
L00(p) & \triangleq & \land\ pc[p] = 0 \\
& & \land\ \exists\, \mathit{new\_ln} \in \{1, 4\} :\ \land\ pc' = [pc\ \text{EXCEPT}\ ![p] = \mathit{new\_ln}] \\
& & \qquad\qquad\qquad\qquad\ \land\ \text{CASE}\ \mathit{new\_ln} = 1 \to \exists\, \mathit{new\_v} \in \mathit{PosInts} : v' = [v\ \text{EXCEPT}\ ![p] = \mathit{new\_v}] \\
& & \qquad\qquad\qquad\qquad\qquad\ \square\quad \mathit{new\_ln} = 4 \to v' = v \\
& & \land\ \text{UNCHANGED}\ \langle L,\ Q,\ i,\ j,\ l,\ x,\ T\rangle
\end{array}
$$

$$
\begin{array}{lll}
L01(p) & \triangleq & \land\ pc[p] = 1 \\
& & \land\ pc' = [pc\ \text{EXCEPT}\ ![p] = 2] \\
& & \land\ i' \ \ = [i\ \ \text{EXCEPT}\ ![p] = L] \\
& & \land\ L' \ \ = L + 1 \\
& & \land\ \text{UNCHANGED}\ \langle Q,\ j,\ l,\ x,\ v\rangle \\
& & \land\ T' \ \ = \{u \in \mathit{TupleDomain} : \\
& & \qquad\ \land\ \mathit{tTypeOK}(u)' \\
& & \qquad\ \land\ \exists\, t \in T : \\
& & \qquad\qquad \exists\, S \in \text{SUBSET}\ \mathit{UnlinearizedEnqs}(t)' : \\
& & \qquad\qquad\ \exists\, \mathit{seq} \in \mathit{Perm}(S) : \\
& & \qquad\qquad\qquad \land\ u.\mathit{state} = \mathit{Concat}(t.\mathit{state}, [k \in \text{DOMAIN}\ \mathit{seq} \mapsto v[\mathit{seq}[k]]]) \\
& & \qquad\qquad\qquad \land\ \forall\, q \in \mathit{ProcSet} : \text{IF}\ q \in S \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN}\ u.\mathit{res}[q] = \mathit{ACK} \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE}\ u.\mathit{res}[q] = t.\mathit{res}[q]\}
\end{array}
$$

$$
\begin{array}{lll}
L02(p) & \triangleq & \land\ pc[p] = 2 \\
& & \land\ pc' = [pc\ \text{EXCEPT}\ ![p] = 3] \\
& & \land\ Q' = [Q\ \ \text{EXCEPT}\ ![i[p]] = v[p]] \\
& & \land\ \text{UNCHANGED}\ \langle L,\ i,\ j,\ l,\ x,\ v,\ T\rangle
\end{array}
$$

$$
\begin{array}{lll}
L03(p) & \triangleq & \land\ pc[p] = 3 \\
& & \land\ pc' = [pc\ \text{EXCEPT}\ ![p] = 0] \\
& & \land\ T' = \mathit{Filter}(p,\ \mathit{ACK}) \\
& & \land\ \text{UNCHANGED}\ \langle L,\ Q,\ i,\ j,\ l,\ x,\ v\rangle
\end{array}
$$

$$
\begin{array}{lll}
L04(p) & \triangleq & \land\ pc[p] = 4 \\
& & \land\ pc' = [pc\ \text{EXCEPT}\ ![p] = 5] \\
& & \land\ l' \ \ = [l\ \ \text{EXCEPT}\ ![p]\ = L] \\
& & \land\ \text{UNCHANGED}\ \langle L,\ Q,\ i,\ j,\ x,\ v,\ T\rangle
\end{array}
$$

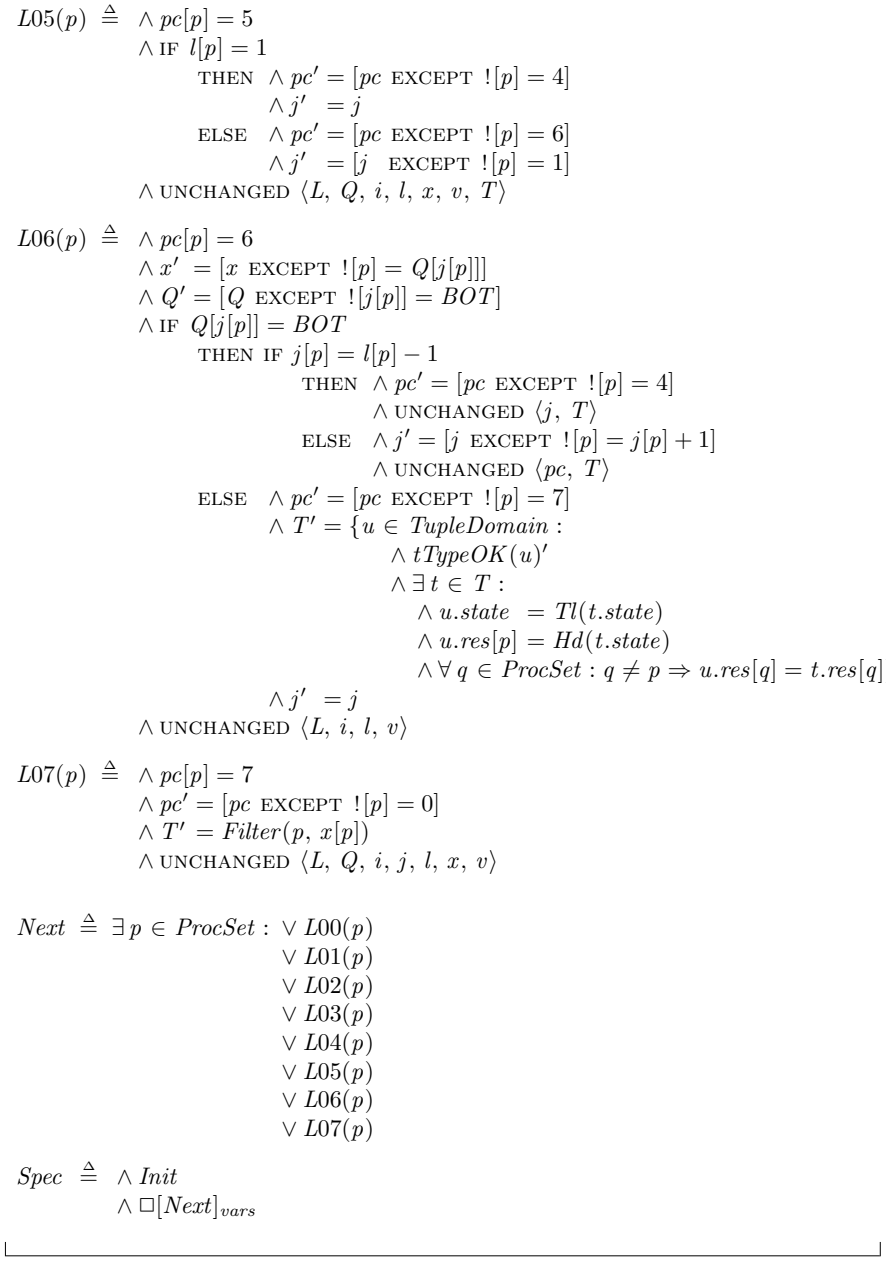

Figure 6: The TLA+ specification of the transformation

References

[Afe+93] Yehuda Afek et al. "Atomic Snapshots of Shared Memory". In: *J. ACM* 40.4 (Sept. 1993), pp. 873–890. ISSN: 0004-5411. DOI: 10.1145/153724.153741.

[AZ20] Jeremy Avigad and Richard Zach. "The Epsilon Calculus". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Fall 2020. Metaphysics Research Lab, Stanford University, 2020. URL: https://plato.stanford.edu/archives/fall2020/entries/epsilon-calculus/.

[BDD07] Richard Bonichon, David Delahaye, and Damien Doligez. "Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs". In: *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, Proceedings*. Ed. by Nachum Dershowitz and Andrei Voronkov. Vol. 4790. Lecture Notes in Computer Science. Springer, 2007, pp. 151–165. DOI: 10.1007/978-3-540-75560-9_13.

[Cha+08] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. "A TLA$^+$ Proof System". In: *Proceedings of the LPAR Workshops, CEUR Workshop*. Vol. 418. Aug. 2008, pp. 17–37. URL: https://www.microsoft.com/en-us/research/publication/tla-proof-system/.

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. "Introduction to Algorithms, Third Edition". In: The MIT Press, 2009. Chap. 10. Elementary Data Structures. ISBN: 0262033844.

[GR22] Valentin Goranko and Antje Rumberg. "Temporal Logic". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2022. Metaphysics Research Lab, Stanford University, 2022. URL: https://plato.stanford.edu/archives/sum2022/entries/logic-temporal/.

[Her91] Maurice Herlihy. "Wait-Free Synchronization". In: *ACM Trans. Program. Lang. Syst.* 13.1 (Jan. 1991), pp. 124–149. ISSN: 0164-0925. DOI: 10.1145/114005.102808.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972.

[Int22] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z. Apr. 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[Jay05] Prasad Jayanti. "An Optimal Multi-Writer Snapshot Algorithm". In: *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*. STOC '05. Baltimore, MD, USA: Association for Computing Machinery, 2005, pp. 723–732. ISBN: 1581139608. DOI: 10.1145/1060590.1060697.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 032114306X.

[MB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer

Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[Mer08] Stephan Merz. "The Specification Language TLA$^+$". In: *Logics of Specification Languages*. Ed. by Dines Bjørner and Martin C. Henson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 401–451. ISBN: 978-3-540-74107-7. DOI: 10.1007/978-3-540-74107-7_8.

[Mic15] Microsoft Research-Inria Joint Center. *TLA+ Proof System: Tutorial*. (Accessed: 8 May 2022). 2015. URL: https://tla.msr-inria.inria.fr/tlaps/content/Documentation/Tutorial.html.

[Pet83] Gary L. Peterson. "Concurrent Reading While Writing". In: *ACM Trans. Program. Lang. Syst.* 5.1 (Jan. 1983), pp. 46–55. ISSN: 0164-0925. DOI: 10.1145/357195.357198.

[PN94] Lawrence Charles Paulson and Tobias Nipkow. *Isabelle: A Generic Theorem Prover*. Lecture Notes in Computer Science. Springer, 1994. ISBN: 9783540582441.

[YV22] Ugur Y. Yavuz and Lizzie Hernandez Videa. *easy-to-verify-linearizability*. GitHub repository. May 2022. URL: https://github.com/uguryavuz/easy-to-verify-linearizability.