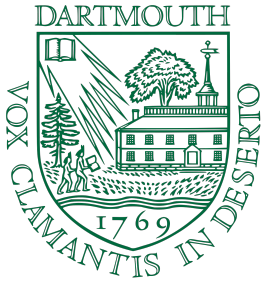
Producing easy-to-verify proofs of linearizability

Ugur Y. Yavuz
Advisor: Prasad Jayanti



A thesis submitted to the Faculty
in partial fulfillment of the requirements for
the degree of Bachelor of Arts in Computer Science

Department of Computer Science
Dartmouth College
June 1, 2021

Contents

Abstract

Proofs of linearizability tend to be complex and lengthy, rendering their verification challenging for readers. We provide a novel technique to produce easy-to-verify proofs of linearizability, with the help of mechanical proof assistants. Specifically, we reduce the task of proving the correctness of a linearizable object implementation, to a proof of an inductive invariant of a slightly modified version of the implementation. As the latter is a task many mechanical proof systems (such as TLAPS) are well-suited to undertake, this reduction allows the verification of the proof by the reader, to only consist of a trivial syntactic check of whether the modification fulfills certain criteria. To demonstrate the practicability of this technique, we provide two applications.

Acknowledgements

I would like to express my deepest gratitude towards my advisor, Dr. Prasad Jayanti, whose support and encouragement have enabled me to explore the fascinating field of concurrent algorithms over the course of this project. I would also like to thank Siddhartha Jayanti and Dr. Stephan Merz, whose help and suggestions were indispensable for the completion of this work. Lastly, I would like to thank my family, as well as my friends at Dartmouth and overseas for their support.

1 Introduction

Our research concerns concurrent, multi-process computing systems with *asynchronous* processes with unique identifiers, whose relative speeds of execution vary arbitrarily over time. These processes communicate via *shared objects* which are *typed* and *atomic*. The *type* of a shared object determines the set of operations that processes can invoke on the object with arguments, and for each operation, specifies how the operation affects the *state* of the shared object, as well as what response is returned when the operation returns to the invoking process. An object is said to be *atomic*, when it is the case that once any process invokes an operation on the object, the operation takes effect instantaneously – i.e., the state changes and the response is returned right away.

The design of concurrent algorithms that satisfy strong safety and liveliness properties is known to be a challenging task, but a task which can be helped by the availability of atomic objects for interprocess communication. For instance, in the x86 architecture, a process can atomically execute *read*, *write*, and *compare-and-swap* operations on memory words [1]. Nevertheless, when designing parallel programs, there is often a need for many more types of atomic objects such as queues and hash tables, which are by default not supported by the hardware. This gap, however, can be bridged by designing algorithms that implement atomic objects of the desired types, using atomic objects that are already available.

An algorithmic implementation of an object specifies a method for each operation that the object supports. For instance, an implementation of a queue object Q provides the code for the methods $enqueue_p(e)$ and $dequeue_p()$, which a process p can invoke to enqueue an element e in Q, or dequeue an element from Q. When p calls and executes such a method, this execution lasts for a duration, dependant on the length of the code comprising the method, as well as the speed at which p executes code. As such, p's operations on Q are not instantaneous, i.e., they are not atomic. However, in a seminal paper published in 1990, Herlihy and Wing identify and define a property called *linearizability* which, once satisfied by an implementation, makes the implemented object behave like an atomic one. In short, this property requires each method to appear to take effect instantaneously at some point (referred to as the *linearization point*) within the interval of time in which the method is executed [2].

Designing linearizable implementations of objects using locks is straightforward: one could specify each method such that the corresponding operation is executed once the lock is acquired, and the lock is released afterwards. The correctness of this implementation follows easily from the use of the lock, which ensures that even if multiple processes attempt to operate on the object concurrently, they modify it one at a time, in the order in which they acquire the lock. However, the use of locks is not well suited for asynchronous systems, because if a process slows down after acquiring a lock, other processes in need of the lock have to wait. To overcome such limitations, researchers have proposed *wait-free* implementations, which guarantee that every process completes each operation in a bounded number of its own steps, regardless of whether other processes are slow or have even possibly stopped taking steps [3].

Implementations that are both linearizable and wait-free are highly desirable, because objects with these properties are qualitatively indistinguishable from atomic objects. These algorithms, however, tend to be complex and are prone to subtle race conditions that fail linearizability. It is therefore imperative that every linearizable algorithm be accompanied by a rigorous proof of its linearizability. This poses a challenge, as such proofs tend to be quite long, and hence the possibility that the proof contains errors is not negligible, potentially masking errors in the algorithm. The sheer length of, as well as the amount of detail within these proofs also render a meticulous verification impractical, if not impossible.

One clear way out of this predicament is to produce a proof of linearizability that is either entirely machine generated, or largely machine generated with only a small component in need of human verification. Furthermore, we want this component to be easy to verify: so easy that the human verifier should be able to attest to its veracity without having to even understand the algorithm in question.

In this paper, we show that this goal is realizable. This assertion follows from our main theorem which states that for any algorithm A, A is linearizable if a certain statement S is an invariant of a *transformation* B of A. This reduction of the question whether A is linearizable to the question whether S is an invariant of B is particularly useful because, with some effort, one can use proof assistants to produce an automated proof that S is an invariant of B. This machine-generated proof, along with our theorem, implies the linearizability of A. Furthermore, a reviewer seeking to verify the correctness of this assertion, only has to check whether B is a valid transformation of A, that is whether B can be obtained from A using only the legal transformation rules allowed by our theorem. This check is a simple syntactic check, therefore a human verification is quick and easy as desired.

We demonstrate the effectiveness of our technique by producing easy-to-verify proofs of linearizability of two algorithms, first a simple implementation of a *read/write* object, then an implementation of a *single-writer*, *single-scanner snapshot* object [4]. We describe a legal transformation for each algorithm, identify its invariants, write the specification for the transformation in TLA$^+$: a formal specification language created by Leslie Lamport [5], and use the TLA$^+$ Proof System (TLAPS) [6] to come up with an easily verifiable proof that S is indeed an invariant of the transformation of the algorithm, which in turn implies the linearizability of the original algorithm.

2 The model

Processes and *shared objects* are the building blocks of *algorithms*. An *implementation* is an algorithm of a certain kind, and a *linearizable implementation* is an implementation satisfying a certain property. In this section, we will provide formal definitions for each of these concepts.

Definition 2.1 (Process)**.** A *process* has a distinct name p and a program Π to execute. Each line of its program Π consists of one or more operations, wherein p invokes an operation on a shared object and gets a response, or alternatively reads or modifies *local variables*. Among the local variables of p, the *program counter* is distinguished, and it contains a line number of a line in Π. This line can be thought of as the line that p will execute next. A *state* of p (or its *local state*) is any assignment of values to p's local variables, including its program counter.

Π may consist of multiple methods. For brevity, we will refer to the i-th line of the method α of the program Π as $\Pi.\alpha.i$, and the program counter of a process p as $p.PC$ throughout the paper.

Definition 2.2 (Object and type)**.** An *atomic shared object*, henceforth simply referred to as an *object*, has a distinct name O and a *type* T. T is a tuple which consists of the following components:

- Σ: a set of states. The *state* of O (O.state) at any time is an element of Σ.
- $\mathcal{P}$: a set of process names. A process p can invoke an operation on the object O if and only if $p \in \mathcal{P}$.
- OP: the set of *operations* supported by the object.
- $OP_p \subseteq OP$ for each process $p \in \mathcal{P}$. p may invoke an operation α on O only if $p \in \mathcal{P}$ and $\alpha \in OP_p$.
- ARG_α for each operation $\alpha \in OP$. Any argument of an operation α invoked on O must be an element of ARG_α.
- RES_α for each operation $\alpha \in OP$. Similarly, any response returned by an operation α invoked on O must be an element of RES_α.
- δ: a relation[1] such that for any state $\sigma \in \Sigma$, $p \in \mathcal{P}, \alpha \in OP_p$ and $arg \in ARG_\alpha$, there exists at least one pair $(\sigma', res) \in \Sigma \times RES_\alpha$ where $(\sigma', res) \in \delta(\sigma, p, \alpha, arg)$. Intuitively, this means that if an operation α is invoked by process p with the argument arg when the state of O is σ, it is possible for the state of the object to become σ' and for the response res being returned by O.

Here are some examples of object types:

- A *read/write* object O supports the operation *read* which takes no arguments and returns $O.state$, and *write* which takes an argument v to write into $O.state$, and returns an acknowledgement `ack`. There are no specific restrictions on the set of states of the type other than the fact that it should consist of single memory words

[1] If the type of interest is deterministic, which is often the case, δ may be defined as a function such that $\delta(\sigma, p, \alpha, arg) = (\sigma', res)$ instead of being defined as a potentially one-to-many relation.

(it could be the set of integers, natural numbers, booleans, strings etc.), or the set of processes it supports.

The *read/write* type is therefore defined by the following tuple:

$$(\Sigma, \mathcal{P}, \{read, write\}, \{\{read_p, write_p\} : p \in \mathcal{P}\}, \{ARG_{read} := \varnothing, ARG_{write} := \Sigma\}, \\ \{RES_{read} := \Sigma, RES_{write} := \{\texttt{ack}\}\}, \delta)$$

where $\delta(\sigma, p, read, \varnothing) = (\sigma, \sigma)$ and $\delta(\sigma, p, write, v) = (v, \texttt{ack})$.

- A *read/CAS* object O supports the operation *read* which takes no arguments and returns $O.state$, and CAS (shorthand for *compare-and-swap*) which takes two arguments x and v, which sets $O.state$ to v and returns *true* if $O.state = x$, or leaves $O.state$ untouched and returns $false$ if $O.state \neq x$. There are no specific restrictions on the set of states of the type other than the fact that it should consist of single memory words, or the set of processes it supports.

  The *read/CAS* type is therefore defined by the following tuple:

  $$(\Sigma, \mathcal{P}, \{read, CAS\}, \{\{read_p, CAS_p\} : p \in \mathcal{P}\}, \{ARG_{read} := \varnothing, ARG_{CAS} := \Sigma \times \Sigma\}, \\ \{RES_{read} := \Sigma, RES_{CAS} := \{true, false\}\}, \delta)$$

  where $\delta(\sigma, p, read, \varnothing) = (\sigma, \sigma)$ and

  $$\delta(\sigma, p, CAS, (x, v)) = \begin{cases} (v, true) & \text{if } x = \sigma \\ (\sigma, false) & \text{if } x \neq \sigma \end{cases}.$$

- An *n-process, single-writer, single-scanner snapshot object* supports $n+1$ processes $p_1, \ldots, p_n$, and a special process s, consists of n components (words), and supports two operations: a process p_i can call *write*, which takes an argument v, writes v into the i-th component and returns an acknowledgement $\texttt{ack}$; and the process s can call *scan*, which does not take any arguments, and returns the sequence of the values of all components [4]. Importantly, the object operates under the assumption that there are no concurrent write operations to any one component.

  Each component is a single memory word, and they all hold values of the same type – let D refer to the set of possible values for each component. The *n-process, single-writer, single-scanner snapshot* type is therefore defined by the tuple:

  $$(\Sigma := D^n, \{s\} \cup \{p_i : 1 \leqslant i \leqslant n\}, \{scan, write\}, \{scan_s\} \cup \{\{write_{p_i}\} : 1 \leqslant i \leqslant n\}, \\ \{ARG_{scan} := \varnothing, ARG_{write} := D\}, \{RES_{scan} := D^n, RES_{write} := \{\texttt{ack}\}\}, \delta)$$

  where $\delta(\sigma, s, scan, \varnothing) = (\sigma, \sigma)$; $\delta(\sigma, p_i, write, v) = ((\sigma_1, \ldots \sigma_{i-1}, v, \sigma_{i+1}, \ldots, \sigma_n), \texttt{ack})$.

Definition 2.3 (Algorithm)**.** An *algorithm* is a specification of the following:

- $\mathcal{P}$: the set of processes for which the algorithm will specify a program.
- Π_p for each process $p \in \mathcal{P}$. Π_p is the program for process p.
- $Init_p$ for each process $p \in \mathcal{P}$, each one a non-empty subset of p's states. Intuitively, this means that each element of $Init_p$ can be the initial state of p.

- $\mathcal{O}$: a set of object names; and a type for every object $O \in \mathcal{O}$. Objects in $\mathcal{O}$ are referred to as the *base objects* of the algorithm, and their types are called *base types*.
- $Init_O$ for each object $O \in \mathcal{O}$, each a non-empty subset of $T.\Sigma$ where T is O's type. Intuitively, this means that each element of $Init_O$ can be O's initial state.

An *algorithm* must also satisfy the following integrity condition: if $p \in \mathcal{P}$ invokes an operation α on an object O in any line of Π_p, it must be the case that $O \in \mathcal{O}$ and $\alpha \in T.OP_p$ (i.e. the type of O supports p's execution of α).

Definition 2.4 (Runs and histories)**.** Let A be an arbitrary algorithm. Let us first define some concepts which will motivate the definition of a *run* and a *history* of the algorithm A:

- A *configuration* of A is any assignment of states to the processes and objects of A.
- An *initial configuration* of A is a configuration where the state of each process p of A is in $Init_p$, and the state of each object O is in $Init_O$.
- An *event* of A is any pair (p, a) such that p is any process of A, and a is any line of code in the program Π_p.
- A *step* of A is a triple $(C, (p, a), C')$ where C is any configuration of A, p is any process of A, a is the line of code in Π_p that is pointed to by the program counter of p in C, and C' is the configuration that results from p's execution of a from configuration C.

A *run* of A is a finite sequence of the form $C_0, (p_1, a_1), C_1, (p_2, a_2), \ldots, C_k$, or an infinite sequence of the form $C_0, (p_1, a_1), C_1, (p_2, a_2), \ldots$ such that C_0 is an initial configuration of A; and for all i, $(C_{i-1}, (p_i, a_i), C_i)$ is a step of A.

A *history* of A is the subsequence containing all of the events in a run of A. For a run $C_0, (p_1, a_1), C_1, (p_2, a_2), \ldots$ of A, $(p_1, a_1), (p_2, a_2)$ is the history which corresponds to the run.

Definition 2.5 (Implementation)**.** Let T be a type defined by the following tuple: $(\Sigma, \mathcal{P}, OP, \{OP_p : p \in \mathcal{P}\}, \{ARG_\alpha : \alpha \in OP\}, \{RES_\alpha : \alpha \in OP\}), \delta)$. An *implementation* of the type T, initialized to a state $\sigma \in \Sigma$, is an algorithm A such that:

- $A.\mathcal{P} = \mathcal{P}$. Intuitively, this means that the algorithm specifies a program for each process in the process set of T.
- For each $p \in \mathcal{P}$, Π_p consists of a main method and for each operation $\alpha \in OP_p$, a method $\alpha_p(arg)$ where $arg \in ARG_\alpha$. Without loss of generality, we assume that the method α_p has a canonical form such that it returns only on its last line of code.
- For each process p, the main method of p (i.e. $\Pi_p.main$) is a loop in which a single call is repeatedly made: p non-deterministically picks an operation $\alpha \in OP_p$ and a corresponding $arg \in ARG_\alpha$, and places a call to $\alpha_p(arg)$.

If A is an algorithm that meets the mentioned criteria, then A is said to implement a type T. This definition of implementation is merely syntactic. We will shortly state a correctness condition for implementations: namely, linearizability.

Definition 2.6 (I/O behavior)**.** Let A be an algorithm that implements a type T. Then, an event of A of the form (p, a) is an *I/O event* if a is either a call to a method or a return from the method. In other words, either a is a call of $\alpha_p(arg)$ where $\alpha \in OP_p, arg \in ARG_\alpha$, or a is "**return** res" for some $res \in RES_\alpha$ for $\alpha \in OP_p$.

An *I/O behavior* of the implementation A is a subsequence of a history of A containing all of the *I/O events* in the history.

Definition 2.7 (Atomic implementation)**.** An *atomic implementation* is a trivial implementation that implements a type T, using an atomic object of the same type.

Formally, an *atomic implementation* of a type T initialized to a state σ, is an implementation of T where only a single base object O of type T (where $O.State = \sigma$ in the initial configuration) is used, and for each process $p \in T.\mathcal{P}$ and operation $\alpha \in T.OP_p$, the code for the method $\alpha_p(arg)$ where $arg \in ARG_\alpha$ only consists of the following two lines.

method $\alpha_p(arg \in ARG_\alpha)$
1. $res \leftarrow O.\alpha(arg)$
2. **return** res

Definition 2.8 (Linearizable implementation)**.** A *linearizable implementation of type* T is any implementation A of T such that every I/O behavior of A is an I/O behavior of an atomic implementation of T.

3 The main theorem: reduction of linearizability to an invariant

Our theorem is defined on a model of n processes, and it relies on a specific *transformation* B of an implementation A of a type T. We will first explain what this *transformation* consists of.

Definition 3.1 (Transformation)**.** Let A be an algorithm which implements a type $T = (\Sigma, \mathcal{P}, OP, \{OP_p : p \in \mathcal{P}\}, \{ARG_\alpha : \alpha \in OP\}, \{RES_\alpha : \alpha \in OP\}), \delta)$ initialized to state $\sigma \in \Sigma$.

A *transformation* of A is any algorithm B, where we keep track of the possible states of the abstract object O of type T implemented by A as well as the various ways in which each method any process is executing, could be linearized. Each possible way in which the operations of processes may linearize, corresponds to a possible value for the object state, as well as possible values for the responses that the processes receive.

In B, we keep the record of these possibilities by adding or modifying tuples of length $n+1$ (one for the state, and the remainder are triples containing the operations, arguments and the responses for each process) in a set of tuples Δ. For each process $p \in \mathcal{P}, B.\Pi_p$ is such that $B.\Pi_p$ contains all the lines of $A.\Pi_p$, in addition to an update of the set of tuples Δ at any line. Let us now define B formally.

A *transformation* of A is any algorithm B such that:

- $B.\mathcal{P} = \mathcal{P}$.
- $B.Init_p = A.Init_p$ for any $p \in \mathcal{P}$.
- $B.\mathcal{O} = A.\mathcal{O} \cup \Delta$, where Δ is a set of tuples of length $n+1$, and for any $t \in \Delta$, t has components $t.State$ and $t.p$ for each $p \in \mathcal{P}$, such that $t.State \in \Sigma$ and for each process p:

$$t.p \in \{—\} \cup \left(OP_p \times \bigcup_{\alpha \in OP_p} ARG_\alpha \times \left(\left(\bigcup_{\alpha \in OP_p} RES_\alpha \right) \cup \{\perp\} \right) \right).$$

  - This means that in Δ, each tuple t is such that $t.State$ is the state of an object of type T, and for each process p, $t.p$ is either blank (containing $\perp$ in every component, we will occasionally use — to denote this), or a triple consisting of an operation p can execute on an object of type T, an argument for this operation, and a response that this operation may return (in addition to $\perp$). For non-blank tuples, we will refer to each of these components as $t.p.op$, $t.p.arg$, and $t.p.res$.

- $B.Init_O = A.Init_O$ for any $O \in A.\mathcal{O}$. And $B.Init_\Delta = \{t\}$ where t is a tuple of length $n+1$ such that $t.State = \sigma$, and $t.p = —$ for any process p.
- For each $p \in \mathcal{P}$, each line of $B.\Pi_p$ contains the operations specified in each corresponding line of $A.\Pi_p$, in addition to a potential update of Δ, subject to the following integrity conditions:

– Rule T1: Upon the invocation by some process p of a method corresponding to an operation $\alpha \in OP_p$ with the argument $arg \in ARG_\alpha$, Δ is updated such that it only contains, for tuples t that were in Δ before the invocation where $t.p = —$, an identical tuple t' except with $t'.p$ being set to $(\alpha, arg, \perp)$.

– Rule T2: If the executed line $a \in B.\Pi_p$ is a return statement, Δ is updated such that it only contains, for tuples t that were in Δ before the execution of a where $t.p \neq —$ and $t.p.res$ equals the value that is returned after the execution of a, an identical tuple t' except with $t'.p$ being set to —.

– Rule T3: For all other lines $a \in B.\Pi_p$, $t' \in \Delta$ only if Δ contained a corresponding tuple t prior to the execution, for which there exists a sequence of k distinct processes (potentially of length 0) where p_i denotes the i-th process, such that $t.p_i \neq —$ and $t.p_i.res = \perp$ for each p_i in the sequence, with the following property:

  Let s_0 denote $t.State$ and $(s_1, r_1), (s_2, r_2), \ldots (s_k, r_k)$ be a sequence of pairs from $\Sigma \times \bigcup_{\alpha \in OP} RES_\alpha$ such that for each i such that $1 \leqslant i \leqslant k$:

$$(s_i, r_i) = \delta(s_{i-1}, p_i, t.p_i.op, t.p_i.arg).$$

  t' must be such that $t'.State = s_k$ and for each i such that $1 \leqslant i \leqslant k$, $t'.p_i.res = r_i$. Note that if $k = 0$, then $t' = t$.

Finally, before presenting the main theorem, we will define the notion of an *invariant*.

Definition 3.2 (Invariant)**.** An *invariant* of an algorithm A is any statement that holds true in every configuration of every run of A.

Theorem 3.1 (The main theorem)**.** An implementation A of the type T is linearizable if $\Delta \neq \varnothing$ is an invariant of any transformation B of A.

It is clear that B is an implementation of T, as it supports the same set of processes, and in the program Π_p of each process $p \in \mathcal{P}$, $\Pi_p.main$ continuously invokes methods that correspond to operations supported by T.

Let σ denote the initial state of the object implemented by both A and B. To prove that B is a linearizable implementation of T, we will refer to two lemmas and a corollary. In all three, the implementations A and B, the type T, and the initial state σ are the ones which are referred to in the theorem.

Lemma 3.1. If $\Delta \neq \varnothing$ is an invariant of B, then for each run R of B that ends in a configuration C, for any $t \in C.\Delta$, there exists a run R_a of the atomic implementation of T initialized to σ, that ends in a configuration C_a and satisfies the following conditions:

- The I/O behaviors of R and R_a are identical.
- $t.State = C_a.O.State$, where O is the base object in the atomic implementation.
- If $t.p = —$, then $C.p.PC = C_a.p.PC = \Pi_p.main.1$, where p is an arbitrary process.
- If $t.p \neq —$ and $t.p.res \neq \perp$, then $C_a.p.PC = \Pi_p.(t.p.op).2$.
- If $t.p \neq —$ and $t.p.res = \perp$, then $C_a.p.PC = \Pi_p.(t.p.op).1$.

Proof. This can be proven by induction on the number of steps in a run R of B.

For the base case, consider a run $R = C$, consisting of only an initial configuration (i.e., a run with 0 steps). Let R_a be a run of the atomic implementation which also only consists of an initial configuration C_a. We note that the I/O behaviors of R and R_a are identical as they are simply empty sequences, and that $t.State = C_a.O.State = \sigma$ by initialization. For any process p, $t.p = —$ and in an initial configuration of any implementation of T, $p.PC = \Pi_p.main.1$. Therefore, the claim is correct for the base case.

Assume that the claim holds for runs of B with n steps. Consider a run $R' = C_0, (p_1, a_1), C_1, \ldots, C_n, (p_{n+1}, a_{n+1}), C_{n+1}$ of B with $n+1$ steps. By our inductive hypothesis, we know that for the run $R = C_0, \ldots, C_n$, that 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.
- $t.State = C_a.O.State$.
- For any $p \in \mathcal{P}$, $t.p = — \implies C_n.p.PC = C_a.p.PC = \Pi_p.main.1$.
  $t.p \neq — \wedge t.p.res \neq \perp \implies C_a.p.PC = \Pi_p.(t.p.op).2$.
  $t.p \neq — \wedge t.p.res = \perp \implies C_a.p.PC = \Pi_p.(t.p.op).1$.

To prove the claim for R', we want to show that there exists a run R'_a of the atomic implementation with the desired properties. There are three cases to examine: a_{n+1} is either a method call from the main method, a return statement, or any other internal line of a method.

Assume a_{n+1} is a method call from the main method. By Rule T1, we know that for $t \in C_{n+1}.\Delta$, there exists a $u \in C_n.\Delta$ where $u.p_{n+1} = —$, and $u.State = t.State$ as well as $u.p = t.p$ for all other processes p. Let $R'_a = R_a, (p_{n+1}, a_{n+1}), C'_a$. As (p_{n+1}, a_{n+1}) is an I/O event for both B and the atomic implementation, the I/O behaviors of R' and R'_a are identical. By our inductive hypothesis, we knew that $u.State = C_a.O.State$. Noting that the invocation does not change the state of the base object in R'_a, we can then observe that $t.State = u.State = C_a.O.State = C'_a.O.State$. After a_{n+1}, $C_{n+1}.p.PC = C'_a.p.PC = \Pi_p.(t.p.op).1$, $t.p \neq —$ and $t.p.res = \perp$. Furthermore, the program counters as well as the tuples of all other processes remain unchanged. Therefore, we observe that the claim is correct for this case.

Assume a_{n+1} is a return statement. By Rule T2, we know that for $t \in C_{n+1}.\Delta$, there exists a $u \in C_n.\Delta$ where $u.p_{n+1} \neq —$ and $u.p_{n+1}.res \neq \perp$, and $u.State = t.State$ as well as $u.p = t.p$ for all other processes p. Let $R'_a = R_a, (p_{n+1}, a_{n+1}), C'_a$. As (p_{n+1}, a_{n+1}) is an I/O event for both B and the atomic implementation, the I/O behaviors of R' and R'_a are identical. By our inductive hypothesis, we knew that $u.State = C_a.O.State$. Noting that the return line does not change the state of the base object in R'_a, then we can then observe that $t.State = u.State = C_a.O.State = C'_a.O.State$. After a_{n+1}, $C_{n+1}.p.PC = C'_a.p.PC = \Pi_p.main.1$, and $t.p = —$. Furthermore, the program counters as well as the tuples of all other processes remain unchanged. Therefore, we observe that the claim is correct for this case.

Finally, assume a_{n+1} is any other line of a non-main method. By Rule T3, we know that for $t \in C_{n+1}.\Delta$, there are two possibilities:

- There exists a tuple $u = t \in C_n.\Delta$, meaning that the sequence of processes that used by the rule used has length 0. Let $R'_a = R_a$. As (p_{n+1}, a_{n+1}) is not an I/O event and $t.p_{n+1} \neq —$, it is clear that R'_a is a run with the desired properties.
- There exists a $u \in C_n.\Delta$ such that, for a sequence of processes $q_1, \ldots, q_k$ where $u.q_i.op = t.q_i.op, u.q_i.arg = t.q_i.arg$, and $u.q_i.res = \perp$ and a sequence of states $s_1, \ldots, s_k$, it is the case that:

  $$\delta(u.State, q_1, u.q_1.op, u.q_1.arg) = (s_1, u.q_1.res)$$
  $$\delta(s_{i-1}, q_i, u.q_i.op, u.q_i.arg) = (s_i, u.q_i.res) \text{ for } 2 \leqslant i \leqslant k,$$

  and $t.State = s_k$.

  By the inductive hypothesis, we know that for all q_i, $C_a.q_i.PC = \Pi_{q_i}.(u.q_i.op).1$. Then, let $R'_a = R_a, (q_1, \Pi_{q_1}.(u.q_1.op).1), C'_1, \ldots, (q_k, \Pi_{q_k}.(u.q_k.op).1), C'_k$. Let $C'_0 := C_a$ for consistency of notation. As neither the step added to R to create R', nor any of the steps added to R_a to create R'_a are I/O events, the I/O behaviors of R' and R'_a remain identical.

  Note that the invocation of $O.(u.q_i.op)(u.q_i.arg)$ in $\Pi_{q_i}.(u.q_i.op).1$ from a configuration C'_{i-1} to reach C'_i when $O.state = s_{i-1}$ results in $C'_i.O.State = s_i$. As $u.State = s_0$, $t.State = s_k$, and $u.State = C'_0.O.State$ by the inductive hypothesis, it is also the case that $t.State = C_a.O.State$.

  Finally, after the execution of the events $(q_i, \Pi_{q_i}.(u.q_i.op).1)$ for all $1 \leqslant i \leqslant k$, it is the case that $t.p \neq —$ and $t.p.res \neq \perp$. Furthermore, as for all q_i, $C_a.q_i.PC = \Pi_{q_i}.(u.q_i.op).1$, it is now that case that $C'_k.q_i.PC = \Pi_{q_i}.(t.q_i.op).2$. The program counters, as well as the tuples of all other processes remain unchanged. Therefore, we observe that the claim is correct for this case.

This completes the proof by induction. The claim is hence correct. □

Corollary 3.1.1. *B is a linearizable implementation of T if $\Delta \neq \varnothing$ is an invariant of B.*

Proof. If $\Delta \neq \varnothing$ is an invariant of B, then we know that for each run R of B that there exists a run R_a of the atomic implementation of T such that the I/O behaviors of R and R_a are identical. Then, the I/O behavior of any run of B is also an I/O behavior of the atomic implementation of T. Therefore, by definition, B is a linearizable implementation of T. □

Lemma 3.2. *If B is a linearizable implementation of T, A is also a linearizable implementation of T.*

Proof. For any run R' of B, of the form $C'_0, (p_1, a_1), C'_1, \ldots, (p_n, a_n), C'_k$, there is a corresponding run R of A of the form $C_0, (p_1, a_1), C_1, \ldots, (p_n, a_n), C_n$, where for any variable v other than Δ, we have $C_i.v = C'_i.v$ for any i. Therefore, we can observe that the I/O behaviors of both runs are identical. Hence, if B is a linearizable implementation of T, by the definition of a linearizable implementation, we have that A is also a linearizable implementation of T. □

Proof of Theorem 3.1. The proof follows immediately from Corollary 3.1.1 and Lemma 3.2. □

4 Technique and applications

In this section, we will describe a technique to produce easily verifiable proofs of linearizability for implementations A of any type T, making use of the theorem described in the previous section.

The technique consists of the following steps:

1. Design a legal transformation B of A.
2. State an invariant I of B.
3. Help TLAPS generate a proof that I is an invariant of B.
4. Help TLAPS generate a proof that $I \implies \Delta \neq \varnothing$.

By virtue of Theorem 3.1, the last step implies that A is a linearizable implementation of T. For a reader to verify this proof, it would suffice to verify whether B is indeed a legal transformation of A, which is a simple syntactic check, as all other claims are proven mechanically using the TLAPS proof assistant. Hence, this technique yields a proof of linearizability which is easy-to-verify as desired.

We note that in TLAPS, the standard method to prove that a statement is in fact an invariant of an algorithm, involves finding an *inductive invariant* of the algorithm which implies the desired invariant [7]. An invariant is said to be *inductive*, when its correctness can be proven via mathematical induction. More specifically, an inductive invariant holds true in the initial configuration of the algorithm, and in the configuration that is reached following the execution of any line of code in the algorithm by any process, from an arbitrary configuration in which the invariant is assumed to hold true. If an inductive invariant implies our desired invariant, it means that the statement holds true in any configuration that could be reached by the execution of the algorithm, thus confirming its invariance.

We will now provide two applications of this technique: a proof of linearizability for an implementation of a *read/write* object (using a *read/CAS* object), and another proof of linearizability for an implementation of a *single-writer, single-reader snapshot* object – both of which are types that have been defined in the previous sections. For each type, we will provide the pseudocode for an implementation A of the corresponding type, claimed to be linearizable, a transformation B of the said implementation and an inductive invariant of B which implies $\Delta \neq \varnothing$, whose correctness is mechanically verified in TLAPS.

4.1 A *read/write* object using a *read/CAS* object

We have already defined the types *read/write* and *read/CAS*. For simplicity, let us restrict the state set of the type to natural numbers, and let there be n processes. Let T denote the *read/write* object type. We will provide the pseudocode for an algorithm A, that we claim is a linearizable implementation of T. Note that X below is a *read/CAS* object. We will occasionally refer to $X.State$ as X.

method $main_p()$
0. **while** *true*
   $\alpha \in T.OP_p$, $arg \in T.ARG_\alpha$
   $\alpha_p(arg)$

Initial values
$X, x_p, v_p \in \mathbb{N}$

method $write_p(v_p)$
1. $x_p \leftarrow X.read_p()$
2. $X.\text{CAS}_p(x_p, v_p)$
3. **return** `ack`

method $read_p()$
4. $x_p \leftarrow X.read_p()$
5. **return** x_p

We will prove that A is a linearizable implementation of T, by first describing a legal transformation B of A, then providing an inductive invariant for B verified to be correct by TLAPS and thereby showing that $\Delta \neq \varnothing$ is an invariant of B, which by our main theorem implies that A is linearizable. In the interest of brevity, let $t!p{:}x$ denote the tuple t, except with the value x in the component p. Here is the pseudocode for a transformation B of A.

method $main_p()$
0. **while** *true*
   $\alpha \in T.OP_p$, $arg \in T.ARG_\alpha$
   $\Delta \leftarrow \{t!p{:}(\alpha, arg, \bot) : t \in \Delta\}$
   $\alpha_p(arg)$

method $write_p(v_p)$
1. $x_p \leftarrow X.read_p()$
2. $X.\text{CAS}_p(x_p, v_p)$
   if $X = x_p$
   $\Delta \leftarrow \{u : \wedge u.State = v_p, u.p.res = \texttt{ack}$
   $\wedge \exists t \in \Delta : t.State = x_p, t.p.res = \bot$
   $\wedge \forall q \in \mathcal{P} : t.q.res = ack \vee q.PC \neq 2 \implies u.q.res = t.q.res$
   $t.q.res \neq ack \wedge q.PC = 2 \implies u.q.res \in \{\bot, ack\}\}$
3. **return** `ack`
   $\Delta \leftarrow \{t!p{:}(—) : t \in \Delta, t.p.res = \texttt{ack}\}$

method $read_p()$
4. $x_p \leftarrow X.read_p()$
   $\Delta \leftarrow \{t!p{:}(t.p.op, t.p.arg, X) : t \in \Delta\}$
5. **return** x_p
   $\Delta \leftarrow \{t!p{:}(—) : t \in \Delta, t.p.res = x_p\}$

The operations on Δ when a method is first called, and at Lines 3 and 5 clearly satisfy the definition of a transformation, by rules T1 and T2. At Line 1 and Line 2 (if the CAS is unsuccessful), rule T3 is applied, and the corresponding sequence of processes is empty. At Line 3, rule T3 is applied, and the corresponding sequence only consists of the process executing the line; as such, the operations at these lines also satisfy the definition. At Line 2, if the CAS succeeds, rule T3 is applied and the corresponding sequence of processes for each tuple u at Line 2, is any arbitrary ordering of processes q for which $u.q.res = \texttt{ack}$, with the caveat that p is at the end of the sequence. Therefore, this operation also satisfies the definition. We can then conclude that B is indeed a legal transformation of A.

Here is an inductive invariant of B:

$$
\begin{aligned}
IInv := \; & \land \; \Delta \neq \varnothing \\
& \land \; \forall t \in \Delta : t.State = X \\
& \land \; \exists t \in \Delta : (\forall q \in \mathcal{P} : q.PC = 3 \implies t.q.res = \texttt{ack}) \\
& \land \; \forall p \in \mathcal{P} : p.PC = 1 \implies (\forall t \in \Delta : t.p.res = \bot) \\
& \land \; \forall p \in \mathcal{P} : p.PC = 2 \implies (\exists t \in \Delta : t.p.res = \bot) \\
& \land \; \forall p \in \mathcal{P} : p.PC = 2 \implies (X \neq x_p \implies (\exists t \in \Delta : t.p.res = \texttt{ack})) \\
& \land \; \forall p \in \mathcal{P} : p.PC = 2 \implies (\forall t \in \Delta : t.p.res \in \{\bot, \texttt{ack}\}) \\
& \land \; \forall p \in \mathcal{P} : p.PC = 2 \implies (\forall t \in \Delta : t.p.res = \texttt{ack} \\
& \qquad\qquad \implies (\exists u \in \Delta : u = t!p{:}(t.p.op, t.p.arg, \bot))) \\
& \land \; \forall p \in \mathcal{P} : p.PC = 2 \implies (X \neq x_p \\
& \qquad\qquad \implies (\forall t \in \Delta : t.p.res = \bot \\
& \qquad\qquad\qquad \implies (\exists u \in \Delta : u = t!p{:}(t.p.op, t.p.arg, \texttt{ack})))) \\
& \land \; \forall p \in \mathcal{P} : p.PC = 3 \implies (\exists t \in \Delta : t.p.res = \texttt{ack}) \\
& \land \; \forall p \in \mathcal{P} : p.PC = 4 \implies (\forall t \in \Delta : t.p.res = \bot) \\
& \land \; \forall p \in \mathcal{P} : p.PC = 5 \implies (\forall t \in \Delta : t.p.res = x_p)
\end{aligned}
$$

The specification of B in TLA$^+$, and a mechanically verified TLAPS proof of the inductive invariance of $IInv$ can be found in [8]. TLAPS was able to automatically verify a subset of the conjuncts of the invariant. However, for most of the syntactically complex conjuncts, it required assistance in witness picking. The proof amounts to 1159 lines and is verified in 1.5 minutes on a modern mid-range laptop.

Note that $\Delta \neq \varnothing$ is a conjunct of $IInv$, and therefore the fourth and final step of the technique is trivially completed. Hence, we have provided an easy-to-verify proof that A is a linearizable implementation of T, whose verification would be sufficed by the verification of whether B is a legal transformation of A.

4.2 A *single-writer, single-scanner* snapshot object

We have already defined the type *n-process, single-writer, single-scanner snapshot*. Let us also restrict the state set of the type to natural numbers. We will use T to denote this type, and we will provide the pseudocode for an algorithm A that we claim is a linearizable implementation of T.

We will consider an algorithm first published by Jayanti in 2003 [4]. We note that proving this implementation's linearizability for $n = 1$ is sufficient to prove its linearizability for any n, and this is due to the *single-writer* assumption that there are no concurrent writes on the same component. Therefore, we will assume $n = 1$, and let the single writing process be named w. The following algorithm is a modified version of Jayanti's algorithm in light of this assumption:

method $main_p()$
0. **while** $true$
   $\alpha \in T.OP_p$, $arg \in T.ARG_\alpha$
   $\alpha_p(arg)$

method $write_w(v)$
1. $A \leftarrow v$
2. **if** $X = true$
3. $\quad B \leftarrow v$
4. **return** `ack`

Initial values
$X = false$
$A, a, v \in \mathbb{N}$;
$B, b \in \mathbb{N} \cup \{\perp\}$

method $scan_s()$
5. $X \leftarrow true$
6. $B \leftarrow \perp$
7. $a \leftarrow A$
8. $X \leftarrow false$
9. $b \leftarrow B$
10. **if** $b = \perp$ **return** a **else** b

We will prove that A is a linearizable implementation of T, by first describing a legal transformation B of A, then providing an inductive invariant for B verified to be correct by TLAPS and thereby showing that $\Delta \neq \emptyset$ is an invariant of B, which by our main theorem implies that A is linearizable.

Let any $t = (t_1, t_2, t_3) \in \Delta$ be such that $t.State = t_1, t.w = t_2, t.s = t_3$. Here is the pseudocode for a transformation B of A.

Initial values
$X = false$
$A, a, v \in \mathbb{N}$;
$B, b \in \mathbb{N} \cup \{\perp\}$

method $main_p()$
0. **while** $true$
   $\alpha \in T.OP_p$, $arg \in T.ARG_\alpha$
   $\Delta \leftarrow \{t!p{:}(\alpha, arg, \perp) : t \in \Delta\}$
   $\alpha_p(arg)$

method $\underline{write_w(v)}$

1. $A \leftarrow v$
   if $X = false \vee s.PC = 6 \vee (s.PC = 7 \wedge B = \perp) \vee (B = v) \vee (s.PC = 8 \wedge B = \perp \wedge a = v)$
   $\quad \Delta \leftarrow \{(v, (t.w)!res{:}\texttt{ack}, t.s) : t \in \Delta\}$
   else
   $\quad \Delta \leftarrow \{(v, (t.w)!res{:}\texttt{ack}), t.s) : t \in \Delta\} \cup \Delta$
2. **if** $X = true$
3. $\quad B \leftarrow v$
   $\quad \Delta \leftarrow \{t : t \in \Delta, t.w.res = \texttt{ack}\}$
4. **return** `ack`
   $\Delta \leftarrow \{(t.State, —, t.s) : t \in \Delta, t.w.res = \texttt{ack}\}$

method $\underline{scan_s()}$

5. $X \leftarrow true$
6. $B \leftarrow \perp$
7. $a \leftarrow A$
8. $X \leftarrow false$
   $\Delta \leftarrow \cup \{(v, (t.w)!res{:}\texttt{ack}, (t.s)!res{:}t.State) : t \in \Delta, t.w.res = \perp, w.PC \neq 1\}$
   $\quad \cup \{(t.State, t.w, (t.s)!res{:}t.State) : t \in \Delta, t.w.res = \perp, w.PC = 1\}$
   $\quad \cup \{(t.State, t.w, (t.s)!res{:}t.State) : t \in \Delta, t.w.res = \texttt{ack}\}$
9. $b \leftarrow B$
10. **if** $b = \perp$
    $\quad$ **return** a
    $\quad \Delta \leftarrow \{(t.State, t.w, —) : t \in \Delta, t.s.res = a\}$
    else
    $\quad$ **return** b
    $\quad \Delta \leftarrow \{(t.State, t.w, —) : t \in \Delta, t.s.res = b\}$

This is clearly a legal transformation of A. Lines 2, 5, 6, 7 and 9 contain no updates, so they vacuously follow Rule T3. Invocation of methods, as well as Lines 4 and 10 follow Rules T1 and T2. Lines 1 and 3 follow Rule T3, for a sequence consisting of only w. Finally, Line 8 follows Rule T3 for sequences $(s, w), (s), (s)$ for each set used to build Δ.

Here is an inductive invariant of B:

$$
\begin{aligned}
IInv := \; & \wedge \forall t \in \Delta : t.w.res = \texttt{ack} \implies t.State = A \\
& \wedge w.PC = 1 \implies (\exists t \in \Delta : t.State = A \wedge t.w.res = \bot) \\
& \wedge w.PC = 1 \implies (\forall t \in \Delta : t.State = A \wedge t.w.res = \bot) \\
& \wedge w.PC = 2 \implies A = v \\
& \wedge w.PC = 2 \implies (\exists t \in \Delta : t.State = v \wedge t.w.res = \texttt{ack}) \\
& \wedge w.PC = 2 \implies (X = false \implies (\forall t \in \Delta : t.State = v \wedge t.w.res = \texttt{ack})) \\
& \wedge w.PC = 2 \implies (\exists t \in \Delta : t.w.res = \bot \\
& \qquad\qquad \implies ((s.PC \in \{7,8\} \wedge B \neq \bot \wedge B \neq v) \vee (s.PC = 8 \wedge A \neq a \wedge B = \bot))) \\
& \wedge w.PC = 3 \implies A = v \\
& \wedge w.PC = 3 \implies (\exists t \in \Delta : t.State = v \wedge t.w.res = \texttt{ack}) \\
& \wedge w.PC = 3 \implies (s.PC \in \{9,10\} \implies (\exists t \in \Delta : t.State = v \wedge t.w.res = \texttt{ack} \wedge t.s.res = v)) \\
& \wedge w.PC = 3 \implies (\exists t \in \Delta : t.w.res = \bot \\
& \qquad\qquad \implies ((s.PC \in \{7,8\} \wedge B \neq \bot \wedge B \neq v) \vee (s.PC = 8 \wedge A \neq a \wedge B = \bot))) \\
& \wedge w.PC = 4 \implies A = v \\
& \wedge w.PC = 4 \implies (\exists t \in \Delta : t.State = A \wedge t.w.res = \texttt{ack}) \\
& \wedge w.PC = 4 \implies (\forall t \in \Delta : t.State = A \wedge t.w.res = \texttt{ack}) \\
& \wedge s.PC = 5 \implies X = false \\
& \wedge s.PC = 5 \implies (\forall t \in \Delta : t.s.res = \bot) \\
& \wedge s.PC = 5 \implies (\forall t \in \Delta : t.State = A \wedge (w.PC \neq 1 \implies t.w.res = \texttt{ack})) \\
& \wedge s.PC = 6 \implies X = true \\
& \wedge s.PC = 6 \implies (\forall t \in \Delta : t.s.res = \bot) \\
& \wedge s.PC = 7 \implies X = true \\
& \wedge s.PC = 7 \implies (\forall t \in \Delta : t.s.res = \bot) \\
& \wedge s.PC = 7 \implies (B \neq \bot \implies (\exists t \in \Delta : t.State = B)) \\
& \wedge s.PC = 7 \implies (B = \bot \implies (\exists t \in \Delta : t.State = A)) \\
& \wedge s.PC = 8 \implies X = true \\
& \wedge s.PC = 8 \implies (\forall t \in \Delta : t.s.res = \bot) \\
& \wedge s.PC = 8 \implies (B \neq \bot \implies (\exists t \in \Delta : t.State = B)) \\
& \wedge s.PC = 8 \implies (B = \bot \implies (\exists t \in \Delta : t.State = a)) \\
& \wedge s.PC = 9 \implies X = false \\
& \wedge s.PC = 9 \implies (\forall t \in \Delta : t.s.res \neq \bot) \\
& \wedge s.PC = 9 \implies (B \neq \bot \implies (\exists t \in \Delta : t.s.res = B)) \\
& \wedge s.PC = 9 \implies (B = \bot \implies (\exists t \in \Delta : t.s.res = a)) \\
& \wedge s.PC = 9 \implies (\forall t \in \Delta : t.State = A \wedge w.PC \neq 1 \implies t.w.res = \texttt{ack}) \\
& \wedge s.PC = 10 \implies X = false \\
& \wedge s.PC = 10 \implies (\forall t \in \Delta : t.s.res \neq \bot) \\
& \wedge s.PC = 10 \implies (B \neq \bot \implies (\exists t \in \Delta : t.s.res = b)) \\
& \wedge s.PC = 10 \implies (B = \bot \implies (\exists t \in \Delta : t.s.res = a)) \\
& \wedge s.PC = 10 \implies (\forall t \in \Delta : t.State = A \wedge w.PC \neq 1 \implies t.w.res = \texttt{ack}).
\end{aligned}
$$

The specification of B in TLA$^+$, and a mechanically verified TLAPS proof of the inductive invariance of $IInv$ can be found in [8]. TLAPS was able to automatically verify a subset of the conjuncts of the invariant (for instance, those that pertain to Lines 7 and 8). However, for most of the syntactically complex conjuncts, it required assistance in witness picking. The proof amounts to 2002 lines and is verified in 20 minutes on a modern mid-range laptop.

Note that for each line, we have a conjunct in our invariant which makes an existential statement about elements of Δ, hence $\Delta \neq \varnothing$ is implied by $IInv$, and therefore the fourth and final step of the technique is trivially completed. Hence, we have provided an easy-to-verify proof that A is a linearizable implementation of T, whose verification would be sufficed by the verification of whether B is a legal transformation of A.

5 Conclusion and future work

To summarize, our main theorem has allowed us to reduce the task of proving the linearizability of an implementation, to the task of proving the invariance of $\Delta \neq \varnothing$ for a *transformation* of the implementation, which can be achieved by proving an inductive invariant of the transformation correct in TLAPS. The verification of any proof of linearizability produced using this technique, would only require the verification of the legality of the transformation, which is a simple syntactic check. Therefore, we have successfully accomplished our goal of describing a technique to produce easy-to-verify proofs of linearizability.

Thus far, we have only applied this technique to the problems explained in this thesis. Therefore, in order to further demonstrate the practicability of this technique in a broader selection of cases, we intend to apply this technique to a wider variety of linearizable implementations and verify their linearizability.

References

[1] *Intel® 64 and IA-32 Architectures Software Developer's Manual.* Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z (April 2021). `https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html`

[2] Herlihy, M. P. and Wing, J. M. "Linearizability: a correctness condition for concurrent objects." *ACM Trans. Program. Lang. Syst.* 12, 3, (July 1990), 463–492.

[3] Herlihy, M. "Wait-free synchronization." *ACM Trans. Program. Lang. Syst.* 13, 1, (January 1991), 124–149.

[4] Jayanti, P. "An optimal multi-writer snapshot algorithm." *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC '05)*, (May 2005), 723–732.

[5] Lamport, L. *The TLA$^+$ Hyperbook* (2015). `https://lamport.azurewebsites.net/tla/hyperbook.html`

[6] Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D. and Vanzetto, H. "TLA$^+$ Proofs." *18th International Symposium on Formal Methods – FM 2012*, volume 7436 of *Lecture Notes in Computer Science* (August 2012), 147–154.

[7] Lamport, L. "Using TLC to Check Inductive Invariance." (August 2018). `https://lamport.azurewebsites.net/tla/inductive-invariant.pdf`

[8] Yavuz, U. Y. `easy-to-verify-linearizability`, GitHub (May 2021). `https://github.com/uguryavuz/easy-to-verify-linearizability`