The Distributed Computing Column

Seth Gilbert
National University of Singapore
seth.gilbert@comp.nus.edu.sg

This month, the Distributed Computing Column is featuring Siddhartha Jayanti, winner of the 2023 Principles of Distributed Computing Doctoral Dissertation Award. His thesis presented a multitude of important concurrent algorithms, including the first scalable algorithm for concurrent union-find, algorithms for concurrent fast arrays, new abortable queue locks and recoverable queue locks, and many more. He defined a new fundamental problem known as "generalized wake-up," whose hardness yields new insights on the work needed for a variety of basic objects. And the thesis contains many more results than can be easily summarized in this short paragraph!

This column focuses on one specific part of his thesis: verifying linearizability in concurrent systems. It is a well-known fact that designing correct concurrent algoroithms is incredibly difficult and bugs are rampant. The standard technique for proving that a concurrent data structure is correct is showing that it is *linearizable*. Unfortunately, that too can be quite challenging! In this article, Siddhartha Jayanti develops a new technique for proving linearizability using only "forward reasoning" techniques: there is no need to reason about the future when analyzing the data structure. This technique yields machine-verifiable proofs, and has been applied to a variety of complex wait-free data structures, including snapshot objects and union-find objects.

Overall, then, this column raises the exciting possibility of simple machine-verified proofs for concurrent data structures, and a future containing more correct concurrent data structures!

The Distributed Computing Column is particularly interested in contributions that summarize recent exciting results, propose interesting new directions, or summarize important open problems in areas of interest. If you would like to write such a column, please contact me.

POSSIBILITY TRACKING: A SIMPLE TECHNIQUE FOR MACHINE-VERIFYING LOCK-FREE DATA STRUCTURES

Siddhartha Jayanti
Google Research, USA

1 Introduction

The multicore revolution has ushered in an era of multiprocessor dominance in computing, however, designing efficient concurrent algorithms with iron-clad guarantees of correctness has remained notoriously hard. While a deterministic single-process algorithm has exactly one possible run, due to asynchrony, even a t step concurrent algorithm with just two processes has 2^t possible runs depending on how the steps of the processes interleave. When we consider algorithms with an infinite horizon, even a deterministic concurrent algorithm has uncountably many possible infinite runs. Designing algorithms that are correct in all of these executions is a grueling task, and programmers often fail to account for some of these executions, leading to subtle and dangerous bugs, known as *races*. Races are pernicious, since they can easily be missed in testing but have harsh consequences when deployed in practice. For example:

- **Mars Rover:** a priority inversion bug in its concurrent code crashed the Pathfinder Rover days after its deployment on Mars and jeopardized the entire multi-million dollar NASA space mission Jones [2013].
- **Northeast Blackout of 2003:** a race in the power grid's energy management system stalled the alarm system for an hour, by which time it was too late to stop a cascading electrical outage that affected an estimated 55 million people across eight states of the USA and the province of Ontario, Canada Poulsen [2004].
- **Therac-25:** the software of the radiation therapy machine, Therac-25, suffered from races that caused it to administer radiation doses that were over

[0]Author information: Google Research, Cambridge, MA, USA. *Email:* sjayanti@google.com

a hundred times as potent as the intended dose, which caused the deaths of at least three people and several more injuries Leveson and Turner [1993]; Lim [1998].

Examples of errors in published concurrent data structures are also not left wanting Colvin and Groves [2005]; Doherty [2003]. These illustrations show just how fatal the consequences can be when multiprocessor code is incorrect, and point to the critical need for concurrent algorithms to be furnished with rigorous, machine-verified guarantees of correctness.

1.1 Which algorithms require rigorous, machine-verified guarantees of correctness?

Two foremost guiding principles in the design of software technology are *modularity* and *top-down design*. Together, these principles state that larger applications should be broken down into smaller well-specified components, i.e., methods, data structures, and simpler algorithms, and that these modular components should be developed independently and made efficient so they can in-turn be called and used in several high-level applications. A strength of this prevalent design strategy is that each core modular component can be built and developed in just one place, and the same well-built component can be used freely in a range of applications today and even the unforeseen applications of tomorrow.

The flip-side of this advantage is a corresponding failure mode, which I term *error proliferation*. Namely, if a core component, such as a data structure, has an uncaught error, it runs the risk of being used in innumerable applications and ultimately crashing critical systems. Even if the component was developed with a low-risk application in mind, the principles of software design make it very easy for the same component to later be integrated into a critical application. Thus, all fundamental data structures and algorithms should be considered critical, and rigorous proofs of correctness are indispensable. Since, concurrent data structures and algorithms are particularly hard to reason about, and important execution schedules and subtle races are often missed in pen-and-paper proofs about them, machine-verified guarantees of correctness are critical for them.

In summary, emphasizing machine-certified correctness of low-level modular components, such as data structures, enhances the reliability of several critical high-level applications.

1.2 Concurrent data structures, linearizability, and future dependence

In this column, I focus on machine-verifiable proofs of correctness for concurrent data structures; particularly, on proving *linearizability*. Linearizability Herlihy and Wing [1990] is the long-standing gold standard for concurrent data structure correctness, and it states that data structure operations must appear to take place *atomically*, i.e., instantaneously, at some point between their invocation and return, even in the face of adversarial asynchronous scheduling. The instant in time at which an operation appears to take effect is called its *linearization point*, and the process is said to have *linearized* its operation after that point in time.

Linearizability is a powerful abstraction, since it allows for efficient software implementations which appear atomic even in the face of tremendous concurrency without requiring global data structure locks—solutions include intricate implementations that use fine-grained locking and lightening-fast implementations that are *lock-free* or even *wait-free* Herlihy [1991]. Linearizability also facilitates composability: its horizontal composability property (known as *locality*), allows algorithmists to prove individual implementations correct without worrying about their interactions with other objects; its vertical composability property allows implementors to replace atomic objects with linearizable implementations.

The most intuitive approach for proving linearizability is via *forward reasoning* methods, where the prover reasons about a concurrent data structure by relating its behavior to that of an atomic reference object as time moves forward. In particular, in a *forward simulation* proof Jonsson [1991], the prover keeps a copy of an atomic reference object and performs an induction over the steps of an arbitrary run of an algorithm using the implemented object, and shows that its behavior is identical to that of the algorithm run with the atomic reference object if the reference object performs operations at the linearization points of implemented object. Forward simulation is easiest when each operation linearizes at a particular line in the operation's code, but this proof technique can work even if an operation's linearizes at different lines of code for different calls, or even if a process's operation can linearize at a step of another process executing a different operation. However, forward simulation proofs are only possible when linearization points can be determined only by looking at the past and present, i.e., for the subclass of so called *strongly linearizable* objects Golab et al. [2011].

There are innumerable examples of linearizable objects however, whose linearization points are *future-dependent*. These implementations have the surprising characteristic that every run can be linearized, but the linearization points of operations can depend on what happens in the future. Such linearizable algorithms are traditionally thought of as notoriously hard to machine-verify, since the intuitive forward simulation proof technique cannot be employed on them Jayanti et al.

[2024]. In particular, as the prover inducts over the run, he cannot know when the linearization points occur (since they are future dependent), and therefore cannot simulate the atomic reference object transitions at the time of the linearization points.

Over the past few decades, researchers have furnished some of these future-dependent algorithms with machine-verified proofs using techniques including backward simulation Jonsson [1989], prophecy variables Abadi and Lamport [1991], partial-order maintenance Khyzha et al. [2017], and aspect-oriented proofs Henzinger et al. [2013]. However, each of these techniques is either well known for being complex and unintuitive for algorithmists, or is incomplete and thus requires ad hoc use. Backward simulation is difficult for algorithmists since it requires reasoning backwards in time Vafeiadis [2008]. Prophecy variables require predicting the future, and are often cited as being "difficult to use in practice" Lamport and Merz [2022]. Partial-order maintenance is known to be incomplete Oliveira Vale et al. [2023]; Jayanti et al. [2024]. Finally, aspect-oriented proofs require new theory to develop the aspects of each data type, and thus only a handful of data types are known to be amenable to such proofs Henzinger et al. [2013]; Dodds et al. [2015]; Öhman and Nanevski [2022]. In particular, a simple, sound and complete technique for proving linearizability has eluded researchers until recently.

1.3 A simple, forward reasoning proof technique for linearizability

In the remainder of this column, I describe the *possibility tracking* (a.k.a. *tracking*) technique for proving the linearizability of concurrent data structures Jayanti [2022]; Jayanti et al. [2024]. This technique was originally described in my doctoral dissertation Jayanti [2022] and subsequently published at this year's *ACM Symposium on Principles of Programming Languages (ACM POPL)* Jayanti et al. [2024].

Possibility tracking is universal, sound, and complete. Universality means that tracking can be applied to any data type; soundness means that a data structure implementation can be proved correct by tracking only if it is linearizable, and completeness means that any linearizable implementation can be proved so using tracking. In addition, tracking is simple and intuitive for algorithmists, since it relies only on forward reasoning, and has been used to produce machine-verified proofs of linearizability for future-dependent and widely-used data structures.

The foundation of our idea lies in replacing the single atomic reference object in the forward simulation technique with a set of such atomic reference objects. In particular, we observe that a run of an algorithm exercising a linearizable object may have several possible linearizations. Each of these linearizations may

correspond to a different set of linearization points. While forward simulation maintains just a single atomic reference object, which corresponds to a single possible linearization (i.e., a single possible set of linearization points); our strategy maintains an atomic reference object corresponding to every possible linearization. Since each atomic reference object corresponds to a possible linearization, we call the maintained reference objects *possibilities*. Our proof technique tracks these possibilites over the length of a run, so we call it possibility tracking.

Tracking has been used to give machine-verified proofs of efficient wait-free data structures, including Jayanti's single-writer single-scanner snapshot Jayanti [2005] and the Jayanti-Tarjan union-find objects Jayanti and Tarjan [2016]; Jayanti et al. [2019]; Jayanti and Tarjan [2021], which are used in Google's open-source graph-mining library to enable "parallel clustering algorithms which scale to graphs with tens of billions of edges" Google-Graph-Mining-Team [2023], are the fastest algorithms for computing connected components of large graphs on CPUs Dhulipala et al. [2020] and GPUs Hong et al. [2020], and are employed in several other applications in machine learning Yu et al. [2023]; Wang et al. [2020]; Tseng et al. [2021], graph analysis Shi et al. [2023]; Dhulipala et al. [2020], and program analysis Bloemen et al. [2016].

1.4 Overview

In the remainder of this column, I will describe the tracking method, demonstrate its use in generating a machine-verified proof with a case study of the Herlihy-Wing queue Herlihy and Wing [1987, 1990]—which is notorious in the verification community for its nuanced, future-dependent linearization structure Jung et al. [2019]—and end with some concluding remarks and directions of interest.

A note on proving strong linearizability A variant of our technique, known as Partial Possibility Tracking, is a universal, sound, and complete proof technique for strong linearizability. In fact, our machine-verified proof of the Jayanti-Tarjan union-find objects shows that they are not just linearizable, but strongly linearizable. While I will not say more about strong linearizability in this column, I refer the interested reader to the following reference Jayanti et al. [2024].

2 The Possibility Tracking Proof Technique for Linearizability

To explain how our method works, let $\mathcal{T}$ be any data type, and O be an implementation of type $\mathcal{T}$ for a set Π of processes. To verify that O is linearizable, we

augment O with an auxiliary variable $\mathcal{P}$, which helps track *all* possible linearizations of O. In this augmented implementation, which we shall refer to as O^*, $\mathcal{P}$ is a set of *possibilities*. Each possibility $p \in \mathcal{P}$ is a pair (σ, f). In particular, we define O^* such that a possibility (σ, f) is in $\mathcal{P}$, if and only if, there is a linearization of the run until now which corresponds to O's state being σ, and π's state being $f(\pi)$, for each process $\pi \in \Pi$. That is, for each process $\pi \in \Pi$, $f(\pi)$ states whether π has an ongoing operation on O and if it does, whether that operation has linearized and if it has, what the associated response is. More specifically, $f(\pi)$ holds one of three types of values—$(\perp, \perp, \perp)$, $(op, arg, \perp)$, or (op, arg, res)—with the following meaning. If $f(\pi) = (\perp, \perp, \perp)$, π has no ongoing operation on the implementation O. In the other two cases, π has an ongoing operation op on O with argument arg, i.e., π invoked $op_\pi(arg)$, but the operation has not returned yet. Furthermore, if $f(\pi) = (op, arg, \perp)$, π's operation has not yet linearized and if $f(\pi) = (op, arg, res)$, π's operation has linearized (i.e., has taken effect) with a response of res (but the operation has not yet returned to the caller).

We initialize $\mathcal{P}$ to the singleton set $\{(\sigma_0, f_0)\}$, where σ_0 is the initial state of O and, for all $\pi \in \Pi$, $f_0(\pi) = (\perp, \perp, \perp)$, to reflect that there are no ongoing operations on O, initially. Whenever any process π executes a step, the set $\mathcal{P}$ is updated using the following simple rules:

1. Update on operation invocation: If π calls a method on O to invoke an operation $op(arg)$, each possibility $(\sigma, f) \in \mathcal{P}$ is updated from $f(\pi) = (\perp, \perp, \perp)$ to $f(\pi) = (op, arg, \perp)$, to reflect that $op(arg)$ is invoked, but has not yet linearized.

   Notationally, we denote this transformation to the set of possibilities by $Invoke(\mathcal{P}, \pi, op, arg)$.

2. Update on operation return: If π returns from a method by executing a '**return** r' statement, for each possibility $(\sigma, f) \in \mathcal{P}$, if $f(\pi) = (op, arg, \perp)$ or if $f(\pi) = (op, arg, res)$ and $res \neq r$, then (σ, f) is removed from $\mathcal{P}$; on the other hand, if $f(\pi) = (op, arg, res)$ and $res = r$, then $f(\pi)$ is updated to $(\perp, \perp, \perp)$. The removal in the former case ensures that those atomic configurations that do not reflect what happens in the actual execution are filtered out. The update to $(\perp, \perp, \perp)$ in the latter case reflects that π no longer has an ongoing operation.

   Notationally, we denote this transformation to the set of possibilities by $Filter(\mathcal{P}, \pi, r)$.

3. Update on any step: When π executes any step of a method, $\mathcal{P}$ is updated to reflect the possibility that *any* subset of unlinearized ongoing operations may now linearize in *any* order. Accordingly, suppose that $(\sigma, f) \in \mathcal{P}$ before

π takes the step, k is any non-negative integer, $\pi_1, \pi_2, \ldots, \pi_k$ are distinct processes, $f(\pi_1), f(\pi_2), \ldots, f(\pi_k) = (op_1, arg_1, \bot), (op_2, arg_2, \bot), \ldots, (op_k, arg_k, \bot)$. Furthermore, suppose that, by the specification of the data type $\mathcal{T}$ of O, $r_1, r_2, \ldots, r_k$ are the responses if the operations

$$op_1(arg_1), op_2(arg_2), \ldots, op_k(arg_k)$$

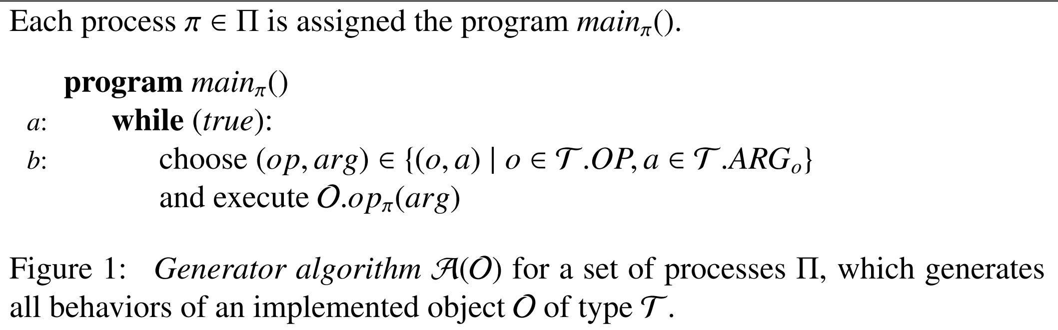
are applied in that order, starting from state σ, and σ' is the state after all operations are applied. Then, after the step, (σ', f') appears in $\mathcal{P}$, where f' is the same as f except that

$$f'(\pi_1), f'(\pi_2), \ldots, f'(\pi_k) = (op_1, arg_1, r_1), (op_2, arg_2, r_2), \ldots, (op_k, arg_k, r_k) .$$

Notationally, we denote this transformation to the set of possibilities by $Evolve(\mathcal{P})$.

A key observation is that at any time t, the set $\mathcal{P}$ contains an atomic configuration (σ, f) if and only if (σ, f) is consistent with *some* legal linearization up to time t. Intuitively, the "only-if" part of the observation is ensured by the second rule which removes a possibility from $\mathcal{P}$ as soon as there is evidence that the atomic configuration is not consistent with a legal linearization of the history. The "if" part is ensured by the third rule which adds to $\mathcal{P}$ *all* possibilities that are consistent with legal linearizations of the history.

Our main theorem is an immediate consequence of the above observation, and it states that any algorithm run on implementation O^* satisfies the invariant $\mathcal{P} \neq \emptyset$ if and only if the implementation O is linearizable. Equivalently, since the *generator algorithm* $\mathcal{A}$ (see Figure 1) exercises an object implementation to produce all of its possible behaviors by repeatedly making idle processes call arbitrary operations, we see that O is linearizable if and only if $\mathcal{P} \neq \emptyset$ is an invariant of $\mathcal{A}(O^*)$.


Each process $\pi \in \Pi$ is assigned the program $main_\pi()$.

program $main_\pi()$
a: **while** (*true*):
b: choose $(op, arg) \in \{(o, a) \mid o \in \mathcal{T}.OP, a \in \mathcal{T}.ARG_o\}$ and execute $O.op_\pi(arg)$

Figure 1: *Generator algorithm* $\mathcal{A}(O)$ for a set of processes Π, which generates all behaviors of an implemented object O of type $\mathcal{T}$.

Theorem 2.1. *Let O be an implementation of an object of type $\mathcal{T}$ initialized to state σ_0 for a set of processes Π, O is linearizable* if and only if *$\mathcal{P} \neq \emptyset$ is an invariant of $\mathcal{A}(O^*)$.*

The theorem gives rise to the possibility tracking verification technique: to verify that an implementation O is linearizable, augment it with the auxiliary variable $\mathcal{P}$ to derive O^* as described, and verify that $\mathcal{P} \neq \emptyset$ is an invariant of $\mathcal{A}(O^*)$. If $\mathcal{P} \neq \emptyset$ is an invariant of $\mathcal{A}(O^*)$, then the theorem implies O is a linearizable implementation of $\mathcal{T}$ and, conversely, if O is a linearizable implementation of $\mathcal{T}$, then the theorem implies that $\mathcal{P} \neq \emptyset$ is an invariant of $\mathcal{A}(O^*)$. Hence, the method is sound and complete. Since the method applies regardless of the data type of the implemented object, it is universal.

3 Case Study: proving the Herlihy-Wing queue

To demonstrate the proof process, I describe our case study of proving the Herlihy-Wing queue Herlihy and Wing [1987, 1990]. I will present the queue implementation in the next subsection, and then explain how we obtained a machine-verified proof of its correctness using possibility tracking.

I chose the Herlihy-Wing queue as the case study, since the algorithm is both short and well known, yet it is notorious in the verification community for being difficult to prove because of its nuanced, future-dependent linearization structure Jung et al. [2019].

Since the Herlihy-Wing queue is nuanced, it is intellectually challenging for a prover to wrap his head around why the algorithm is linearizable. Developing intuition for why the algorithm is linearizable is an inherent part of the proof process, and no proof technique, including possibility tracking can help fully overcome that. Once a prover has understood the intuition for why the queue is linearizable however, possibility tracking makes it easy to translate the intuition into a linearizability proof—even a machine-verified proof; all the prover needs to do is to encode his understanding as a simple algorithmic invariant. That is what I hope to demonstrate to the reader through this case study.

3.1 The Herlihy-Wing Queue Implementation

The Herlihy-Wing queue (see Figure 2) maintains an infinite array of *slots*, i.e., $A[0, 1, 2, \ldots]$, which are initially all *empty*, containing the value $\perp$. The shared counter X stores the value of the next empty slot in A, initially 0. To ENQUEUE an element v_π, process π atomically fetches and increments X (Line 1) to *claim* the next available slot i_π for its enqueue, and simply *places* its element in the claimed slot, thereby *filling* that slot with its element (Line 2) and returns *ack* (Line 3). To DEQUEUE, process π reads the value ℓ_π of the counter X (Line 4), which stores the number of claimed slots, and loops through each index j_π of the array A from 0 to $l_\pi - 1$, *checking* each slot $A[j_\pi]$ and *grabbing* the element in the slot if the slot is

non-empty (Line 5). If π successfully grabs an element, then it returns it (Line 6). Otherwise, if it reaches the end of the loop, π simply tries again.

Note that a dequeue operation only ever returns with an element; that is, it keeps running indefinitely if the queue is empty.

Base Objects:

- X is a read/F&Inc register initialized to 0.
- $A[0, 1, 2, \ldots]$ is an infinite array, where for each $i \in \mathbb{N}$, $A[i]$ is initialized to $\perp$.

```
procedure O.Enqueue_π(v_π)
1:   i_π ← F&Inc(X)
2:   A[i_π] ← v_π
3:   return ack

procedure O.Dequeue_π()
4:   ℓ_π ← X
5:   if ℓ_π = 0 then goto 4 else j_π ← 0
     x_π ← Swap(A[j_π], ⊥)
     if x_π = ⊥ then
         if j_π = ℓ_π − 1 then goto 4
         else { j_π ← j_π + 1; goto 5 }
6:   return x_π
```

Figure 2: Herlihy-Wing queue implementation Herlihy and Wing [1990]. Each numbered line in this implementation O has at most one shared memory instruction, and is performed atomically. The operation $F\&Inc(var)$ is the atomic fetch-and-increment operation, which returns the current value of *var* and increments it by one. The operation $Swap(var, new)$ is the atomic swap operation, which returns the current value of *var* and updates its value to *new*.

Since each enqueue claims a unique slot, and the slots are claimed in order, at first glance, it is tempting to think that the abstract state of the queue will be a tuple of elements order as in the array A. However, this is not the case, since there can be an arbitrary delay between the time an enqueue claims its slot on Line 1 to the time that it actually fills the slot with its element at Line 2. In particular, dequeuers that are looping through the array will go past slots that have been claimed but have not yet been filled, so elements in slots with higher indices can be grabbed before those in lower indices. Furthermore, since dequeuers can be poised at various different parts of the array (i.e, can have different j values), having looped past slots that had been claimed but not yet filled, the order in which elements are dequeued depends heavily on the order in which processes take steps, thus making the linearization of enqueues highly future-dependent. This future-dependence makes the Herlihy-Wing queue notoriously difficult to prove.

3.2 Proving the linearizability of the Herlihy-Wing queue

To show that the implementation O is linearizable via the possibility tracking technique, we must show that the statement $\mathcal{I}_L \equiv (\mathcal{P} \neq \varnothing)$ is an invariant of $\mathcal{A}(O^*)$, where O^* is the possibility tracker presented in Figure 3. We will prove $\mathcal{I}_L$'s invariance by induction over the length of an arbitrary run. $\mathcal{I}_L$ holds in the initial configuration by the tracker definition, so the base case is straightforward. $\mathcal{I}_L$'s validity in subsequent configurations however, relies not only on its validity in the current configuration, but also on the design of the algorithm, i.e., other invariants of the algorithm that capture the states of the various program variables. Thus, in order to go through with the induction, we must strengthen $\mathcal{I}_L$ to a stronger invariant $\mathcal{I}$ that meets two conditions: (a) $\mathcal{I}$ is *inductive* and (b) $\mathcal{I}$ implies $\mathcal{I}_L$.

For most proofs, identifying the strengthened inductive invariant $\mathcal{I}$ is the real intellectual challenge. Once the correct $\mathcal{I}$ is identified, its actual proof by induction tends to be elementary. The statement of the inductive $\mathcal{I}$ for the Herlihy-Wing queue is presented in Figure 4. Since $\mathcal{I}_L$ is a conjunct of $\mathcal{I}$, proving $\mathcal{I}$'s invariance immediately implies $\mathcal{I}_L$'s invariance, and thus that the Herlihy-Wing queue is linearizable.

Understanding the Inductive Invariant

The invariant may appear long, but a closer examination reveals that almost all of the conjuncts—all but $\mathcal{I}_P$—are fairly elementary. $\mathcal{I}_T$ states that slots that are yet to be claimed remain empty. $\mathcal{I}_U$ states that different enqueuing processes claim different slots in the array. $\mathcal{I}_2$ states that a claimed slot remains empty before the process that claims it fills it. $\mathcal{I}_{2,3}$ states that an enqueuer's claimed slot will have an index between 0 and X. Finally, $\mathcal{I}_5$ states that the loop-index j_π will always lie in the loop-interval $[0, \ell_\pi)$ and that the upper loop boundary ℓ_π will never exceed X.

Our core insight about the algorithm is $\mathcal{I}_P$, which identifies a set of possibilities P that must be in the set of tracked possibilities $\mathcal{P}$. The identification of this subset P of "interesting" possibilities is thus the key to understanding why the Herlihy-Wing queue is linearizable. To elucidate this core insight, I explain the definition of P below. Before explaining P, I will explain some preliminaries.

A key insight about the linearization structure of the Herlihy-Wing queue is that while Enqueue operations can linearize at several places in the interval of time between when they grab their slot to when they fill it, a Dequeue operation can always be thought of as linearizing at the moment that it successfully grabs the element that it will return.

Helpful to defining the set of interesting possibilities P, is the function *val*, which maps array indices in $\mathbb{N}$ to their corresponding values. Formally, given an

Base Objects:

- X is a read/F&Inc register initialized to 1.
- $A[0, 1, 2, \ldots]$ is an infinite read/write/Swap array, where each $A[i]$ is initialized to $\bot$.
- $\mathcal{P}$ initialized to $\{(\sigma_0, f_0)\}$ is a meta-configuration, where σ_0 is the empty sequence, and f_0 maps each process $\pi \in \Pi$ to $(\bot, \bot, \bot)$.

```
procedure O*.Enqueue_π(v_π)
    P ← Invoke(P, π, Enqueue, v_π)
1:      i_π ← F&Inc(X)
        P ← Evolve(P)
2:      A[i_π] ← v_π
        P ← Evolve(P)
3:      return ack
        P ← Filter(P, π, ack)
```

```
procedure O*.Dequeue_π()
    P ← Invoke(P, π, Dequeue, ⊥)
4:      ℓ_π ← X
        P ← Evolve(P)
5:      if ℓ_π = 0 then goto 4 else j_π ← 0
        x_π ← Swap(A[j_π], ⊥)
        if x_π = ⊥ then
            if j_π = ℓ_π − 1 then goto 4
            else {j_π ← j_π + 1; goto 5}
        P ← Evolve(P)
6:      return x_π
        P ← Filter(P, π, x_π)
```

Figure 3: Tracker $\mathcal{O}^*$ for the queue implementation $\mathcal{O}$ for processes Π presented in Figure 2

index i, $val(i)$ is defined as: the value of $A[i]$ if $A[i]$ is non-empty, the value of v_π if π has claimed index i, but is yet to fill it, and $\bot$ otherwise.

To understand the definition of P, we ask ourselves the question: *What are the possible states of the queue at any point in time?* We break this question into two parts: understanding which elements are in the queue, and understanding what order they are in.

Firstly, it is clear that the elements in the queue must be of the form $val(i)$ for some indices i. Thus, the set of elements in the queue must be $\{val(i) \mid i \in I, val(i) \neq \bot\}$ for some subset of indices $I \in [0, X)$. Since Enqueue operations linearize by the time they fill their claimed slot, elements that have been placed in the array but have not yet been grabbed must be in the queue, thus $A[i] \neq \bot \implies i \in I$. Elements corresponding to slots that have been claimed but not yet filled may or may not be in the queue, since the corresponding Enqueue operation may or may not have linearized.

After choosing a possible subset of indices I, we get to the what order $\alpha \in Perm(I)$ the corresponding elements $val(i)_{i \in I}$ occupy in the queue state. This is where we make the most incisive insight. We observe that for a given permutation

α, there is a possibility p with state $p.\sigma = val(\alpha_1), \ldots, val(\alpha_{|\alpha|})$ if α is a $Justified$ permutation. Here, we define the predicate $Justified(\alpha)$ to hold if and only if: for every two indices $\alpha_m, \alpha_n \in I$ that appear at the mth and nth position of the permutation α, where $m < n$: either $\alpha_m < \alpha_n$—i.e., the indices are not inverted in the permutation order—or if they are inverted and the former slot $A[\alpha_n]$ is filled, then there must be a dequeing process π that has checked past the smaller index α_n (i.e., $\alpha_n < j_\pi$) in its checking-loop whose upper index ℓ_π exceeds the larger index α_m. In mathematical notation:

$$\begin{aligned} Justified(\alpha) \quad \triangleq \quad & \forall m, n \in [1, |\alpha|] : (\alpha_m < \alpha_n) \lor (A[\alpha_n] \neq \bot \\ & \implies \\ & \exists \pi \in \Pi : pc_\pi = 5 \land \alpha_n < j_\pi \land \alpha_m < \ell_\pi) \end{aligned}$$

Finally, since I is the set of indices corresponding to ENQUEUE operations that have linearized, we know that any pending enequeuers that have linearized are exactly those whose claimed location's indices appear in I. Likewise, since we know that all DEQUEUE operations linearize at the last iteration of Line 5, we know that a pending dequeuer π has linearized if and only if $pc_\pi = 6$. Putting all of these insights together yields the definition of the set of interesting possibilities P, as defined in Figure 4.

The invariant $\mathcal{I}_P$ simply states that this set of interesting possibilities P is non-empty and indeed contained in the set of possibilities $\mathcal{P}$.

Machine-verified proof

With the strengthened invariant $\mathcal{I}$ in hand, the actual induction proof is quite mechanical, making it a perfect fit to be checked and certified by a machine. The proof of the induction step has six cases, one for each line of the implementation, and it comprehensively justifies why each of the invariant conjuncts holds after an arbitrary process π whose program counter currently points to a line l executes that line of code. Our inductive proof of $\mathcal{I}$ is an invariant of $A(O^*)$ has been checked by the TLA+ Proof System (TLAPS) Jayanti et al. [2023b].

4 Related Work

The Linearizability correctness condition for concurrent shared-memory data structures was introduced in a landmark paper by Herlihy and Wing in 1990 Herlihy and Wing [1990]. In the ensuing three decades, a tremendous amount of research has focused on proving linearizability, using various different methods, including: refinements Lamport [1983], forward simulation Jonsson [1991], backward

$$\mathcal{I} \equiv \mathcal{I}_L \wedge \mathcal{I}_P \wedge \mathcal{I}_T \wedge \mathcal{I}_U \wedge \mathcal{I}_2 \wedge \mathcal{I}_{2,3} \wedge \mathcal{I}_5$$

In this expression, the various conjuncts on the right hand side are defined below.

- $\mathcal{I}_L \equiv \mathcal{P} \neq \varnothing$
- $\mathcal{I}_P \equiv P \subseteq \mathcal{P} \wedge P \neq \varnothing$
- $\mathcal{I}_T \equiv \forall i \in \mathbb{N} : i \geq X \implies A[i] = \bot$
- $\mathcal{I}_U \equiv \forall \pi, \pi' \in \Pi : \pi \neq \pi' \wedge pc_\pi, pc_{\pi'} \in \{2, 3\} \implies i_\pi \neq i_{\pi'}$
- $\mathcal{I}_2 \equiv \forall \pi \in \Pi : pc_\pi = 2 \implies A[i_\pi] = \bot$
- $\mathcal{I}_{2,3} \equiv \forall \pi \in \Pi : pc_\pi \in \{2, 3\} \implies 0 \leq i_\pi < X$
- $\mathcal{I}_5 \equiv \forall \pi \in \Pi : pc_\pi = 5 \implies 0 \leq j_\pi < \ell_\pi \leq X$

- $\forall i \in \mathbb{N} : val(i) \triangleq \begin{cases} A[i], & \text{if } A[i] \neq \bot \\ v_\pi, & \text{if } \exists \pi \in \Pi : pc_\pi = 2 \wedge i_\pi = i \\ \bot & \text{otherwise} \end{cases}$

- $Justified(\alpha) \triangleq$
  $\forall m, n \in [1, |\alpha|] : (\alpha_m < \alpha_n) \vee (A[\alpha_n] \neq \bot$
  $\implies$
  $\exists \pi \in \Pi : pc_\pi = 5 \wedge \alpha_n < j_\pi \wedge \alpha_m < \ell_\pi)$

- $$P \triangleq \left\{ p \;\middle|\; \begin{array}{l} \exists I \subseteq [0, X), \exists \alpha \in Perm(I) : \\ \quad \forall i \in I : val(i) \neq \bot \wedge \\ \quad \forall i \in [0, X), A[i] \neq \bot \implies i \in I \wedge \\ \quad Justified(\alpha) \wedge \\ \quad p.\sigma = val(\alpha_1), \ldots, val(\alpha_{|\alpha|}) \wedge \\ \quad \forall \pi \in \Pi : \\ \qquad pc_\pi \in \{1, 2, 3\} \implies p.f(\pi).op = \text{ENQUEUE} \wedge p.f(\pi).arg = v_\pi \\ \qquad pc_\pi \in \{4, 5, 6\} \implies p.f(\pi).op = \text{DEQUEUE} \wedge p.f(\pi).arg = \bot \\ \qquad pc_\pi = 3 \vee (pc_\pi = 2 \wedge i_\pi \in I) \implies p.f(\pi).res = ack \wedge \\ \qquad pc_\pi = 6 \implies p.f(\pi).res = v_\pi \wedge \\ \qquad pc_\pi \notin \{3, 6\} \implies p.f(\pi).res = \bot \end{array} \right\}$$

Figure 4: Invariant $\mathcal{I}$ of $\mathcal{A}(O^*)$, where O^* is the implementation of the queue tracker in Figure 3.

simulation Jonsson [1989], forward-backward simulation Lynch and Vaandrager [1995], history and prophecy variables Owicki and Gries [1976]; Abadi and Lamport [1991], aspect-oriented proofs Henzinger et al. [2013], partial-order maintenance Khyzha et al. [2017], and the use of several proof-logics such as interval temporal logic Schellhorn et al. [2011], separation logic Jung et al. [2019], and category theory based methods Oliveira Vale et al. [2023]. The various techniques differ in range of applicability, mechanization, simplicity of use, scope for modularity and several other quantitative and qualitative metrics. The full body of work is too large to cover in a related work section like this one, but I make an attempt to cover some central ideas here. Several of the well-established techniques are mentioned in Dongol and Derrick's survey paper Dongol and Derrick [2014].

A significant portion of linearizability proofs are simulation proofs. A simulation proof incrementally relates the behavior of an implementation to the behavior of an abstract specification: a *forward simulation* does so in the natural direction of execution, while a *backward simulation* does so in the reverse direction. Forward simulation involves only forward reasoning and is thereby among the most intuitive methods for proving linearizability. Traditionally, forward simulation has been used to prove the linearizability of data structures with fixed linearization points Abdulla et al. [2017]; Vafeiadis [2009]; Amit et al. [2007]. Schellhorn et al. proved that backward simulation is a universal, sound, and complete proof technique for linearizability verification Schellhorn et al. [2014], and gave a mechanized proof in KIV Reif et al. [1998] of the correctness of the Herlihy-Wing queue, which is notorious for its future-dependent linearization points. Backward simulation, however, is not a silver bullet. Backward simulation proofs are famously complex and are generally unintuitive to algorithm designers since they require reasoning about the execution of the algorithm in reverse Vafeiadis [2008]; Dongol and Derrick [2014]; Khyzha et al. [2017]. The simulation techniques can also be combined in a forward-backward simulation Lynch and Vaandrager [1995]; Colvin et al. proved the linearizability of Heller et al.'s concurrent list-based set implementation using this technique Colvin et al. [2006]; Heller et al. [2006]; their proof is verified by the PVS proof system.

Some recent works extend forward simulation like techniques to produce pen-and-paper proofs of linearizability of more complex data structures through the use of "commitment points". In particular, Khyzha et al. give a proof technique that maintains a partial order over operations, such that all total orders that respect the partial order are valid linearizations Khyzha et al. [2017]. This maintenance of a partial order allows them to be more tolerant to future-dependence than traditional forward simulation methods which maintain a single total order. In particular, as the future unfolds, the technique makes the partial order stricter at *commitment points* to eliminate total orders that are no longer linearizable. Khyzha et al. give pen-and-paper proofs for the Herlihy-Wing queue, time-stamped queue,

and an optimistic set. Bouajjani et al. similarly extend forward simulation techniques to show some queues with fixed linearization points for dequeue and some stack data structures can be proved using forward simulation like methods using commitment points and partial orders Bouajjani et al. [2017]. These authors provide pen-and-paper proofs for the Herlihy-Wing queue and a time-stamped stack data structure. Both these works extend the scope of forward reasoning methods beyond data structures with fixed linearization points. However, the commitment points method with a partial order is not complete Oliveira Vale et al. [2023]; Jayanti et al. [2024].

An alternative to simulation based proofs are proofs using history and prophecy variables. History variables, a.k.a. auxiliary variables, remember the past Owicki and Gries [1976], while prophecy variables foresee/predict the future Abadi and Lamport [1991]. Lynch notes that arguments using history variables alone are akin to forward simulation arguments, while those using prophecy variables alone are akin to backward simulation arguments, and those using a combination of history and prophecy variables are akin to forward-backward simulation arguments Lynch and Vaandrager [1995]. Similarly to backward simulation, prophecy variables also suffer from being “difficult to use in practice” Lamport and Merz [2022]. In the context of our paper, the most related work that uses these variables is the Ph.D. thesis of Vafeiadis Vafeiadis [2008]. In particular, he presents a technique that annotates algorithms with single assignment variables, and stores linearization information into the single-assignment variables with the aid of prophecy variables to help in resolve future-dependent linearization points. He uses the technique to obtain machine-verified proofs of a stack, list, RDCSS (restricted double-compare single-swap Harris et al. [2002]), and MCAS (multiword compare-and-swap). This technique however, is restricted to a class of lock-free algorithms that linearize at CAS operations and another class of read-only methods.

Introduced by Henzinger et al. in 2013, aspect-oriented proofs are non-simulation based techniques that exploit the semantics of particular data types in order to reduce proofs of linearizability to proofs of simpler properties called aspects Henzinger et al. [2013]. The technique is inherently non-universal however, requiring new theory to be developed about the aspects that need to be proved about each data type. Henzinger et al.’s original paper develops the theory for queues, and Chakraborty et al. produced a machine-verified proof of the Herlihy-Wing queue using this method Chakraborty et al. [2015]. The technique was later extended for stacks by Dodds et al. [2015]. Recently, the technique has been extended to snapshot objects by Öhman et al. who have also used it to prove several of Jayanti’s snapshot algorithms Jayanti [2005]; Öhman and Nanevski [2022].

Researchers have also explored several specific-purpose program logics for proving linearizability Vafeiadis et al. [2006]; Schellhorn et al. [2011]; Jung et al. [2019]; Oliveira Vale et al. [2023]. Jung et al., in particular, have machine-verified

the linearizability of the Herlihy-Wing queue using the Iris framework for separation logic in Coq Jung et al. [2018]. None of these techniques are known to be complete.

In context, our tracking technique is, to our knowledge, the only forward reasoning method to achieve universality, soundness, and completeness.

5 Conclusion and Remarks

In this ongoing era of the multicore revolution, concurrent algorithms are playing a pivotal role in critical systems. The human mind struggles to tackle the complexities of asynchrony, so traditional pen-and-paper proofs—which often gloss over cases or try to capture the high-level at the cost of missing fine details—are often insufficient to fully convince ourselves that key concurrent algorithms are race-free. The lack of rigorous correctness guarantees of concurrent code have often contributed to the failures of consequential systems, such as the Mars Rover failure, the Northeast Blackout of 2003, and the Therac-2 tragedies. All of this evidence points to the importance of machine-verifying concurrent data structures, the key building blocks of concurrent and parallel algorithms.

While a simple, universal, and complete technique for proving the correctness of concurrent data structures has eluded researchers for decades, recent work has broken this barrier. In this column, I explained the *possibility tracking technique* for proving the linearizability of concurrent data structures, and have demonstrated the technique's efficacy by presenting the machine-verified proof of the notriously challenging Herlihy-Wing queue. The technique has also been used to machine-verify efficient and widely used data structures, including Jayanti's single-writer single-scanner snapshot object, and the Jayanti-Tarjan union-find objects. Collectively, these verified algorithms are noted for their complexity, speed, and wide-spread use.

My principle motivation in writing this column is to share my excitement for machine-verification and attaining reliable guarantees of correctness for distributed algorithms. I look forward to machine-verifiying many more algorithms myself, but am also hoping to see machine-verification of algorithms become more mainstream across the distributed computing community.

Due to the inherent complexity of concurrent algorithms, I believe that machine verification can play an even wider role in providing robust, trusted guarantees. My collaborators and I are extending the possibility tracking technique to incorporate other variants of linearizability, such as *strict linearizability* Aguilera and Frølund [2003], *durable linearizability* Izraelevitz et al. [2016], and *recoverable linearizability* Berryhill et al. [2015]; Jayanti et al. [2023a]. We are also designing techniques to verify liveness properties, such as lock- and wait-

freedom, and properties of mutual exclusion locks, such as starvation-freedom and first-come-first-served fairness. Finally, we are developing techniques to produce machine-verified proofs of time complexity guarantees of multiprocess algorithms.

While linearizability and its variants have become the gold standard for data structure correctness, there are several algorithms both in the literature and in applications that satisfy weaker consistency guarantees, such as sequential and causal consistency. To my knowledge, universal and complete techniques for these data structures are still wanting, and it would be great to see progress on these important directions.

Acknowledgements

I would like to thank my collaborators on the possibility tracking verification work, particularly, Prasad Jayanti and Ugur Yavuz.

References

M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. ISSN 0304-3975.

P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. *International Journal on Software Tools for Technology Transfer*, 19(5):549–563, Oct 2017. ISSN 1433-2787.

M. K. Aguilera and S. Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, Hewlett-Packard Labs, 2003.

D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 477–490. Springer, 2007.

R. Berryhill, W. M. Golab, and M. Tripunitara. Robust shared objects for non-volatile main memory. In E. Anceaume, C. Cachin, and M. G. Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, volume 46 of *LIPIcs*, pages 20:1–20:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

V. Bloemen, A. Laarman, and J. van de Pol. Multi-core on-the-fly SCC decomposition. In *Proceedings of the 21st ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '16, page to appear, 2016.

A. Bouajjani, M. Emmi, C. Enea, and S. O. Mutluergil. Proving linearizability using forward simulations. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II 30*, pages 542–563. Springer, 2017.

S. Chakraborty, T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. *Logical Methods in Computer Science*, Volume 11, Issue 1, Apr. 2015.

R. Colvin and L. Groves. Formal verification of an array-based nonblocking queue. In *10th International Conference on Engineering of Complex Computer Systems (ICECCS 2005), 16-20 June 2005, Shanghai, China*, pages 507–516. IEEE Computer Society, 2005.

R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In T. Ball and R. B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 475–488. Springer, 2006.

L. Dhulipala, C. Hong, and J. Shun. ConnectIt: A framework for static and incremental parallel graph connectivity algorithms, 2020.

M. Dodds, A. Haas, and C. M. Kirsch. A scalable, correct time-stamped stack. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 233–246, New York, NY, USA, 2015. Association for Computing Machinery.

S. Doherty. Modelling and verifying non-blocking algorithms that use dynamically allocated memory. In *Victoria University of Wellington*, 2003.

B. Dongol and J. Derrick. Verifying linearizability: A comparative survey. *CoRR*, abs/1410.6268, 2014.

W. M. Golab, L. Higham, and P. Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In L. Fortnow and S. P. Vadhan, editors, *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 373–382. ACM, 2011.

Google-Graph-Mining-Team. Google graph-mining. https://github.com/google/graph-mining, 2023.

T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 265–279, London, UK, UK, 2002. Springer-Verlag.

S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In J. H. Anderson, G. Prencipe, and R. Wattenhofer, editors, *Principles of Distributed Systems*, pages 3–16, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In P. R. D'Argenio and H. Melgratti, editors, *CONCUR 2013 – Concurrency Theory*, pages 242–256, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1): 124–149, January 1991. ISSN 0164-0925.

M. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 13–26. ACM Press, 1987.

M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. ISSN 0164-0925.

C. Hong, L. Dhulipala, and J. Shun. Exploring the design space of static and incremental graph connectivity algorithms on GPUs. *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, September 2020.

J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In C. Gavoille and D. Ilcinkas, editors, *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, volume 9888 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2016.

P. Jayanti. An optimal multi-writer snapshot algorithm. In H. N. Gabow and R. Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory*

of Computing, Baltimore, MD, USA, May 22-24, 2005, pages 723–732. ACM, 2005.

P. Jayanti, S. Jayanti, and S. Jayanti. Durable algorithms for writable LL/SC and CAS with dynamic joining. In R. Oshman, editor, *37th International Symposium on Distributed Computing (DISC 2023)*, volume 281 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:20, Dagstuhl, Germany, 2023a. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

P. Jayanti, S. Jayanti, U. Y. Yavuz, and L. Hernandez Videa. Artifact for "A Universal, Sound, and Complete Forward Reasoning Technique for Machine-Verified Proofs of Linearizability", POPL 2024, Oct. 2023b.

P. Jayanti, S. Jayanti, U. Yavuz, and L. Hernandez. A universal, sound, and complete forward reasoning technique for machine-verified proofs of linearizability. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024. doi: 10.1145/3632924. URL https://doi.org/10.1145/3632924.

S. Jayanti, R. E. Tarjan, and E. Boix-Adserà. Randomized concurrent set union and generalized wake-up. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC ’19, page 187–196, New York, NY, USA, 2019. Association for Computing Machinery.

S. V. Jayanti. *Simple, Fast, Scalable, and Reliable Multiprocessor Algorithms*. PhD thesis, Massachusetts Institute of Technology (MIT), Department of Electrical Engineering and Computer Science, November 2022. Code available at: https://github.com/visveswara/machine-certified-linearizability.

S. V. Jayanti and R. E. Tarjan. A randomized concurrent algorithm for disjoint set union. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC ’16, pages 75–82, New York, NY, USA, 2016. ACM.

S. V. Jayanti and R. E. Tarjan. Concurrent disjoint set union. *Distributed Comput.*, 34(6):413–436, 2021.

M. Jones. What really happened to the software on the Mars Pathfinder spacecraft? https://www.rapitasystems.com/blog/what-really-happened-software-mars-pathfinder-spacecraft, July 2013.

B. Jonsson. On decomposing and refining specifications of distributed systems. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop,*

Mook, The Netherlands, May 29 - June 2, 1989, Proceedings, volume 430 of *Lecture Notes in Computer Science*, pages 361–385. Springer, 1989.

B. Jonsson. Simulations between specifications of distributed systems. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR '91, 2nd International Conference on Concurrency Theory, Amsterdam, The Netherlands, August 26-29, 1991, Proceedings*, volume 527 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 1991.

R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

R. Jung, R. Lepigre, G. Parthasarathy, M. Rapoport, A. Timany, D. Dreyer, and B. Jacobs. The future is ours: Prophecy variables in separation logic. *Proc. ACM Program. Lang.*, 4(POPL), Dec. 2019.

A. Khyzha, M. Dodds, A. Gotsman, and M. Parkinson. Proving linearizability using partial orders. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings 26*, pages 639–667. Springer, 2017.

L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

L. Lamport and S. Merz. Prophecy made simple. *ACM Trans. Program. Lang. Syst.*, 44(2):6:1–6:27, 2022.

N. Leveson and C. Turner. An investigation of the Therac-25 accidents. *Computer*, 1993.

J. Lim. An engineering disaster: Therac-25, 1998.

N. A. Lynch and F. W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

J. Öhman and A. Nanevski. Visibility reasoning for concurrent snapshot algorithms. *Proc. ACM Program. Lang.*, 6(POPL), Jan. 2022.

A. Oliveira Vale, Z. Shao, and Y. Chen. A compositional theory of linearizability. *Proc. ACM Program. Lang.*, 7(POPL), Jan. 2023.

S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

K. Poulsen. Software bug contributed to blackout. *SecurityFocus*, 2004.

W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. *Automated Deduction—A Basis for Applications: Volume II: Systems and Implementation Techniques*, pages 13–39, 1998.

G. Schellhorn, B. Tofan, G. Ernst, and W. Reif. Interleaved programs and rely-guarantee reasoning with ITL. In C. Combi, M. Leucker, and F. Wolter, editors, *Eighteenth International Symposium on Temporal Representation and Reasoning, TIME 2011, Lübeck , Germany, September 12-14, 2011*, pages 99–106. IEEE, 2011.

G. Schellhorn, J. Derrick, and H. Wehrheim. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. Comput. Logic*, 15(4), September 2014.

J. Shi, L. Dhulipala, and J. Shun. Parallel algorithms for hierarchical nucleus decomposition, 2023.

T. Tseng, L. Dhulipala, and J. Shun. Parallel index-based structural graph clustering and its approximation. In G. Li, Z. Li, S. Idreos, and D. Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1851–1864. ACM, 2021. doi: 10.1145/3448016.3457278. URL https://doi.org/10.1145/3448016.3457278.

V. Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.

V. Vafeiadis. Shape-value abstraction for verifying linearizability. In N. D. Jones and M. Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*, pages 335–348. Springer, 2009.

V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, page 129–136, New York, NY, USA, 2006. Association for Computing Machinery.

Y. Wang, Y. Gu, and J. Shun. Theoretically-efficient and practical parallel DBSCAN. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of*

Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, pages 2555–2571. ACM, 2020. doi: 10.1145/3318464.3380582. URL https://doi.org/10.1145/3318464.3380582.

S. Yu, J. Engels, Y. Huang, and J. Shun. Pecann: Parallel efficient clustering with graph-based approximate nearest neighbor search, 2023.