On the Limits of Automated Linearizability Verification for Modern Concurrency Patterns*

Durwasa Chakraborty
Indian Institute of Technology Madras
Department of Computer Science
India
cs24d011@cse.iitm.ac.in

Abstract

Modern multicore systems operate under relaxed memory models, where the observable behavior of concurrent programs may diverge significantly from sequentially consistent executions. While many existing verification techniques for concurrent data structures assume sequential consistency, practical systems and language runtimes expose weaker memory semantics, thereby complicating formal reasoning about correctness.

This report investigates the problem of verifying correctness properties of concurrent algorithms under different memory models, with particular emphasis on atomicity, linearizability, and progress guarantees. Rather than treating verification as a secondary concern following algorithm design, we position it as the primary challenge in reasoning about modern concurrent systems.

As a concrete vehicle for this study, we examine multi-word compare-and-swap (hereinafter *MCAS*) [4] algorithms, which serve as a representative and demanding case study for modern verification techniques. We survey existing verification methodologies and tools, and identify open challenges in reasoning about linearizability under language-level memory models.

1 Introduction

In contemporary computing systems, performance improvements are achieved primarily through the effective utilization of multicore architectures, rather than through continued increases in single-core clock frequency. This architectural shift follows the breakdown of Dennard scaling [2], under which further increases in transistor density could no longer be accompanied by proportional reductions in power consumption and heat dissipation. Consequently, processor design has evolved toward multicore architectures as the principal mechanism for performance scaling.

As concurrency has become ubiquitous, the central challenge has shifted from achieving parallelism to reasoning about correctness in its presence. The nondeterminism inherent in concurrent execution, together with the exponential growth of possible interleavings on multicore systems, renders conventional testing methodologies that involves an array of testing suites inadequate. Establishing correctness therefore requires formal reasoning frameworks capable of expressing and verifying properties such as atomicity, linearizability, and progress.

The remainder of this report is organized as follows. Section 2 presents the formal definition of linearizability[5] . Section 3 surveys automated and semi-automated approaches to linearizability verification, with particular emphasis on mechanically assisted techniques. [6].Building on this background, Section 4 investigates linearizability in the context of a non-trivial concurrent data structure, namely *kCAS*.[3] This section highlights why traditional linearization-point-based reasoning becomes inadequate in these settings.To address these challenges, we then discuss modern reasoning frameworks based on concurrent separation logic. In particular, we examine recent advances such as Zoo, which provides a compositional framework for verifying fine-grained concurrent algorithms, as well as Cosmo, a concurrent separation logic tailored to the multicore OCaml memory model. TODO.

2 Linearizability: A Correctness Condition for Concurrent Objects[5]

2.1 Motivation

Concurrency is now a given rather than a design choice, and shared *concurrent objects* are unavoidable in system construction. The remaining challenge is no longer how to exploit concurrency, but how to ascribe precise meaning to object behavior in the presence of arbitrary interleavings. The challenge is how to exploit concurrency and assign precise meaning to object behavior in the presence of interleavings, that may grow astronomically. In the battle between performance and correctness, the canons of sound engineering practices always should prioritize correctness, thus motivating the study of correctness conditions for concurrent objects.

The fundamental question concurrency raises is: what is the intended behavior of an object when its operations are interleaved in many possible ways? If thread A performs m operations and thread B performs n operations, the number of possible interleavings grows combinatorially, on the order of $\binom{m+n}{m}$. Whether such executions are correct depends entirely on the object's specification.

*The use of a prepositional opening, often reads as non-assertive. However, this is deliberate: the document is written in reported speech, and this stylistic choice is adopted from Adam Smith's *The Wealth of Nations*.

For example, a queue is not tied to a single behavior; instead it comes with a specification that describes which input/output traces are allowed. In a classic FIFO, queue, if elements are enqueued as $\langle 1, 2, 3\rangle$, every legal execution must dequeue them as verbatim $\langle 1, 2, 3\rangle$. In contrast, large-scale asynchronous systems such Kafka-based queues typically adopt a different delivery specification(s). For example *at-least-once delivery*, in such a system producing $\langle 1, 2, 3\rangle$ may legally result in consumer-visible traces like $\langle 1, 1, 1, 2, 3\rangle$ or multiple replays of $\langle 1, 2, 3\rangle$ $\langle 1, 2, 3\rangle$. Correctness, therefore, is a function of the chosen delivery specification.

This leads to a central question: who defines correctness, and how can it be verified? Linearizability answers this question by providing a precise and compositional correctness condition that allows programmers to reason about concurrent executions using familiar sequential semantics.

2.2 Definition of Linearizability

A history H is *linearizable* if it can be extended to a history H' such that:

1. complete(H') is equivalent to a legal sequential history S,
2. the real-time order is preserved, i.e., $<_H \subseteq <_S$.

Here, complete(H') denotes the maximal subsequence of H' obtained by removing pending invocations. Linearizability permits nondeterminism: multiple sequential histories may justify the same concurrent execution, provided that at least one such history exists.

2.3 Proposed Solution

Consider an execution history H consisting of invocation and response events observed over time. We assume that events are totally ordered by real time, in the sense that no two events occur at exactly the same instant; any two events are separated by some (possibly very small) time difference.

Such a history admits two complementary views. At the lowest level, H reflects the concrete execution of the program: the individual instructions that realize each method call. We refer to this as the *representation-level* view (REP). At a higher level, the same execution can be viewed in terms of method invocations and responses governed by an abstract object specification; this is the *abstract-level* view (ABS).

Linearizability asks whether the behavior observed at the representation level can be explained by some legal abstract execution. Rather than reasoning directly about all possible interleavings, the paper suggests a different perspective. One may consider the set of all sequential executions obtained by linearizing the concrete history $H \restriction_{\mathsf{REP}}$, and ask whether each such execution admits a valid abstract explanation.

Formally, suppose there exists an invariant I over representation states and a mapping

$$A : \mathsf{REP} \longrightarrow 2^{\mathsf{ABS}}$$

that associates each concrete state with abstract state.

The abstraction function maps each concrete state to a set of abstract states consistent with the specification. If, for all $r \in \mathrm{Lin}(H \restriction_{\mathsf{REP}})$, the invariant $I(r)$ holds and

$$A(r) \subseteq \mathrm{Lin}(H \restriction_{\mathsf{ABS}}),$$

then the concrete implementation is linearizable with respect to the abstract object.

This perspective, implicit in the original formulation of linearizability, provides a natural bridge to later work that seeks to mechanize such reasoning and to check these conditions automatically.

3 Automatically Proving Linearizability [6]

3.1 Motivation

We established in the previous section that *linearizability* is the standard correctness condition for concurrent objects, yet establishing it for realistic implementations remains difficult. We explore an alternate way of proving linearizability by identifying *linearization points*: an instant during the operations's execution at which the effect is deemed to occur atomically. While this approach is effective for simple implementations, it becomes increasingly fragile as algorithms grow more sophisticated. In many cases, a linearization point may be conditional, may occur in a different thread, or may depend on future execution. This is particularly common for operations that do not logically modify the shared abstract state, such as unsuccessful lookups or dequeue operations that return EMPTY. For such operations, insisting on a concrete linearization point often obscures, rather than clarifies, the correctness argument.

At the same time a different category of operations exist, *effectful operations*, different in the sense that they necessarily update the abstract state, and those that are *pure*, in the sense that they do not update the state. Effectful operations often, thus by established definition of the point where the effect takes place, becomes a natural candidate for linearization points, while pure operations do not. The key insight of the paper is that these two classes of executions should be treated differently, and that linearizability can be established without explicitly committing to a linearization point for every operation.

3.2 Proposed Solution

The proposed approach replaces explicit reasoning about linearization points and sequential histories with an instrumentation-based verification strategy. Rather than attempting to construct or guess a global linearization, the method embeds the abstract specification directly into the concrete implementation through additional state and assertions. Linearizability is then reduced to the problem of showing that certain assertions cannot fail.

The core idea is to distinguish, within a single execution, whether an operation has performed an effectful abstract update or whether it has remained pure. For effectful executions, the abstract operation is executed at a candidate instruction that is already present in the implementation, such as a successful compare-and-swap. For pure executions, the method refrains from choosing a specific linearization point; instead, it records which return values are consistent with the abstract specification at some point during the execution.

This distinction is realized by instrumenting each operation with auxiliary variables that track abstract effects. One variable records the result of an effectful abstract execution, if such an execution occurs. In parallel, a family of flags records whether particular return values are admissible for a pure execution according to the specification. Initially, no abstract effect is assumed to have taken place, and no return value is assumed to be valid.

As the concrete execution proceeds, candidate effectful linearization points trigger the execution of the abstract specification, and the result of this execution is stored. Independently, whenever the specification permits a pure operation to return a certain value without modifying the abstract state, the corresponding admissibility flag is enabled. At each return point of the concrete method, a single assertion is checked: either an effectful abstract update has occurred and the concrete return value matches the recorded abstract result, or no effectful update has occurred and the returned value is among those permitted by the specification.

Crucially, this assertion localizes the linearizability argument. Instead of quantifying over all possible linearizations of the execution history, the proof obligation reduces to a safety property of the instrumented program. If the assertion holds on all executions, then every concrete behavior admits a valid abstract explanation, and the implementation is linearizable.

3.3 Example: Concurrent Queue

The intuition behind the method can be illustrated using a concurrent queue with operations *enqueue* and *dequeue*. Successful enqueue operations are effectful: they necessarily extend the abstract queue. In typical implementations, this effect corresponds to a concrete instruction that links a new node into the data structure. At this instruction, the abstract enqueue operation is executed, and its result is recorded.

In contrast, a dequeue operation that returns EMPTY does not modify the abstract queue. Such an execution may span multiple reads and checks, and no single instruction need correspond to its logical effect. Rather than assigning it an artificial linearization point, the method simply checks whether returning EMPTY is consistent with the abstract queue being empty at some point during the execution. If so, the return is deemed admissible.

3.4 Verification Procedure

Once the implementation has been instrumented, linearizability checking reduces to verifying that the return-point assertions are never violated. This task can be discharged using existing static analysis techniques, such as abstract interpretation or symbolic execution. Importantly, the verification is performed on the concrete program augmented with ghost state, and does not require explicit enumeration of concurrent interleavings or sequential histories.

3.5 Conclusion

The approach presented in this paper offers a practical and conceptually clean path to automated linearizability proofs. By separating effectful and pure executions and embedding the abstract specification directly into the implementation, it avoids the need for explicit linearization points in cases where they are difficult or unnatural to identify. Linearizability is reduced to a local safety property, enabling the use of mature verification tools.

The method has been implemented in the Cave verification tool and evaluated on a range of concurrent stacks, queues, and set implementations. These results demonstrate that the approach is expressive enough to handle realistic concurrent data structures while remaining amenable to automation.

4 Efficient Multi-word Compare and Swap[3]

4.1 Motivation

At this stage, the verification landscape includes several automated linearizability provers and checkers, supporting both stateless and stateful exploration of concurrent executions.

However, a critical class of implementations remains fundamentally outside the reach of existing automation. In these implementations, linearization points are non-local: the operation may be completed by a thread other than the one that invoked it; the point at which the operation takes effect may lie outside the scope of the method call.

Multi-word compare-and-swap (hereafter MCAS) occupies precisely this gap. MCAS is designed for multicore systems, where performance is achieved not through coarse-grained mutual exclusion, but through cooperative progress, descriptor-based indirection, and helping. As a result, MCAS serves as a compelling and necessary case study to study beyong canonical data structures.

4.2 Proposed Solution

CAS is a hardware-supported atomic primitive that compares the contents of a memory location with an expected value and, if they match, atomically updates it to a new value.

MCAS generalizes CAS to multiple memory locations. An MCAS operation over k locations atomically performs the following logical action:

If all locations $x_1, \ldots, x_k$ contain values $o_1, \ldots, o_k$, then update them to $n_1, \ldots, n_k$; otherwise, leave all locations unchanged.

4.2.1 Classical Descriptor-Based MCAS.

Traditional lock-free MCAS implementations use a descriptor to coordinate the update across multiple locations. The protocol proceeds in three phases.

Example. Consider three memory locations:

$$x = 1, \quad y = 2, \quad z = 3$$

We wish to atomically perform:

$$(x, y, z) : (1, 2, 3) \rightarrow (4, 5, 6)$$

Here, $k = 3$.

Phase 1: Freeze (Lock) Phase :: k CAS operations. Each target location is CASed from its expected value to a pointer to a shared descriptor D. Each CAS both checks the value and locks the location.

$$\texttt{CAS}(x, 1, \&D)$$
$$\texttt{CAS}(y, 2, \&D)$$
$$\texttt{CAS}(z, 3, \&D)$$

If any CAS fails, the operation aborts.

Phase 2: Decision Phase :: 1 CAS operation. The descriptor's status is atomically updated to record the outcome:

$$\texttt{CAS}(D.\text{status}, \text{ACTIVE}, \text{SUCCESS})$$

This CAS constitutes the linearization point of the MCAS operation.

Phase 3: Cleanup Phase :: k CAS operations. Each frozen location is updated based on the descriptor's final state:

$$\texttt{CAS}(x, \&D, 4)$$
$$\texttt{CAS}(y, \&D, 5)$$
$$\texttt{CAS}(z, \&D, 6)$$

The number of CAS operations are:

$$k \text{ (freeze)} + 1 \text{ (decision)} + k \text{ (cleanup)} = 2k + 1 \qquad (1)$$

For the example above, the result is 7 CAS operations.

4.3 Efficient MCAS: $(k + 1)$ CAS

The dominant source of this overhead is the cleanup phase, which requires rewriting each memory location after the outcome of the MCAS has been determined. Rachid et al. observe that this phase is conceptually unnecessary: once the global decision has been made, the final value of each location is already fixed at the logical level.

The central insight of the Rachid et al. protocol is the following:

Rather than restoring or overwriting memory locations after the decision, encode the final outcome in the descriptor and interpret each location's value through the descriptor.

Consequently, a memory location that still references a descriptor is treated as logically containing either the old value or the new value, depending solely on the descriptor's finalized state.

We illustrate the protocol using the same running example:

$$x = 1, \quad y = 2, \quad z = 3$$

The intended atomic update is:

$$(x, y, z) : (1, 2, 3) \rightarrow (4, 5, 6)$$

Phase 1: Freeze Phase :: k CAS Operations. Each target location is atomically updated from its expected value to a pointer to a shared descriptor D:

$$\texttt{CAS}(x, 1, \&D)$$
$$\texttt{CAS}(y, 2, \&D)$$
$$\texttt{CAS}(z, 3, \&D)$$

As in classical MCAS, each CAS simultaneously checks the expected value and marks the location as participating in the ongoing MCAS.

Phase 2: Decision Phase :: 1 CAS Operation. The outcome of the operation is finalized by atomically updating the descriptor's status:

$$\texttt{CAS}(D.\text{status}, \text{ACTIVE}, \text{SUCCESS})$$

This operation serves as the linearization point for the MCAS.

Elimination of the Cleanup Phase. In contrast to classical MCAS, the efficient MCAS [3] performs no explicit cleanup CAS's.

- After the decision phase, memory locations may continue to physically reference the descriptor.
- When a thread reads such a location, it consults the descriptor:
  - If the descriptor state is `SUCCESS`, the location is interpreted as holding the new value.
  - If the state is `FAILED`, the location is interpreted as holding the original value.
- As a result, the memory is logically updated without requiring immediate physical rewrites.

The total number of CAS operations is therefore:

$$k \text{ (freeze)} + 1 \text{ (decision)} = k + 1 \qquad (2)$$

For the three-location example, this reduces the cost from 7 CAS operations to 4 CAS operations.

4.4 Conclusion

MCAS is therefore included not as an isolated or exceptional case, but as a deliberate benchmark for expressiveness. Its correctness fundamentally depends on non-local linearization and helping-driven interference.

5 Zoo: A Framework for the Verification of Concurrent OCaml 5 Programs using Separation Logic[1]

5.1 Motivation

So far, we have focused on linearizability for data structures in which the linearization point can be identified either within the operation itself or within another method invoked during its execution. We now shift attention to concurrent libraries operating under a language- specific memory model. The release of OCaml 5 introduced true parallelism via multicore support, enabling a growing ecosystem of concurrent libraries such as Saturn, Eio, and Kcas. While this substantially broadened OCaml's applicability, it also raised new verification challenges, particularly the lack of a practical framework for reasoning about realistic concurrent OCaml programs that rely on features such as algebraic data types and mutable records.

Existing Iris-based approaches, including *HeapLang*, are expressive but poorly aligned with OCaml. They lack essential language constructs, most notably algebraic data types and mutually recursive functions, and require substantial manual translation from OCaml into the modeling language. This translation burden complicates proof maintenance and weakens the connection to source programs. Moreover, OCaml's concurrency support exposed semantic gaps: atomic record fields were absent, unsafe casts were common, and the semantics of physical equality for compare-and-set were under-specified.

The paper is motivated by these limitations. Its goal is to develop a practical verification framework for real-world concurrent OCaml programs, while refining the language and its semantics to better support safe and verifiable concurrency.

5.2 Proposed Solution

The authors propose Zoo, a comprehensive framework for verifying concurrent OCaml 5 programs using Iris, a state-of-the-art concurrent separation logic mechanized in the Rocq proof assistant.

ZooLang. At the core of Zoo is ZooLang, a language designed to faithfully model a substantial fragment of OCaml. ZooLang supports algebraic data types, mutable and immutable records, references, atomic operations, mutual recursion, and concurrency primitives. It is deeply embedded in Rocq and comes with a formally defined operational semantics and a corresponding Iris-based program logic. The framework includes a tool, ocaml2zoo, which translates OCaml source programs into ZooLang code embedded in Rocq. Unlike HeapLang, which introduces various encodings in the translation that make the relation between source and verified programs difficult to maintain, ZooLang is syntactically very close to OCaml, ensuring that verification artifacts remain aligned with real implementations.

Specifications and Proofs. Once translated to ZooLang, users can write specifications and prove them in Iris. For example, the specification of `stack_push` is:

```
Lemma stack_push_spec t ℓ v :
    <<< stack_inv t ℓ
        | ∀∀ vs, stack_model t vs >>>
      stack_push t v @ ↑ℓ
    <<< stack_model t (v :: vs)
        | RET (); True >>> .
```

As in Hoare logic, the specification consists of pre- and postconditions, each split into a private part (the stack invariant `stack_inv t`) and an atomic part (the abstract stack state `stack_model`). The atomic conditions capture the linearization point, stating that during execution the stack's abstract state is atomically updated from `vs` to `v :: vs`.

Language Extensions: Atomic Record Fields. The authors identified limitations in OCaml's support for atomic operations, particularly the inefficiency of atomic references that introduce additional indirection. To address this, they designed and implemented atomic record fields, allowing individual record fields to be marked as atomic.

Physical Equality: A Precise Semantics. A major technical contribution is a new, precise semantics for physical equality and `compare_and_set`. Physical equality is essential for lock-free algorithms but is subtle and dependent on compiler optimizations such as sharing and unsharing. The authors show that equating physical equality with structural equality is unsound. Instead, they model it as nondeterministic with carefully specified guarantees. They also introduce a mechanism for controlling unsharing through generative constructors, enabling sound reasoning.

5.3 Conclusion

This paper introduces Zoo, a practical framework for verifying concurrent OCaml 5 programs, bridging real-world code and mechanized verification through ZooLang. It also contributes language improvements such as atomic record fields and a precise semantics for physical equality. Using Zoo, the authors verified a subset of the OCaml standard library, components of the Eio library, and a large portion of the Saturn lock-free data structure library, including stacks, queues, bags, and a work-stealing deque. These results demonstrate the practicality and scalability of the framework.

6 Related Work

7 Discussion and Open Problems

8 Conclusion

References

[1] Clément Allain and Gabriel Scherer. Zoo: A framework for the verification of concurrent ocaml 5 programs using separation logic. *Proc. ACM Program. Lang.*, 10(POPL), January 2026.

[2] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. Design of ion-implanted mosfets

with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(3):256–268, 1974.

[3] Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. Efficient multi-word compare-and-swap. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 221–232, Shenzhen, China, 2013. ACM.

[4] Timothy L. Harris and Keir Fraser. A practical multi-word compare-and-swap operation. *Distributed Computing*, 15(1):257–271, 2002.

[5] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(2):463–492, 1990.

[6] Viktor Vafeiadis. Automatically proving linearizability. In *Proceedings of the 22nd International Conference on Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 450–464, Edinburgh, Scotland, 2010. Springer.