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

The transition to multicore architectures has made concurrent programming a necessity rather than an optimization. While linearizability provides a precise correctness condition for concurrent objects, automatically verifying fine-grained lock-free algorithms remains challenging due to interleavings, complex atomic primitives, and weak memory behavior.

This report examines the progression from formal specifications of concurrent correctness, to automated reasoning techniques, to mechanized verification of realistic lock-free data structures, and finally to abstractions that support compositional concurrency under relaxed memory models. Through this layered study, we highlight the central tension between expressive concurrent algorithms and the rigor required to prove them correct. The overarching theme is that scalable concurrency demands not only sophisticated synchronization primitives, but verification frameworks and abstractions capable of reasoning across semantic, algorithmic, and memory-model boundaries.

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.

This report is organized as follows. We begin by recalling the formal definition of linearizability [5] in §2 seminal work by Heerlihy and Wing, the canonical correctness condition for concurrent objects.

§3 surveys automated approaches to linearizability, focusing on the work of Victor Vafeiadis [9], which demonstrates how linearization points and interference reasoning can be mechanized.

We then study Zoo [1],§4, by Allain Clément and Gabriel Scherer, a framework for verifying concurrent OCaml5 programs using separation logic. Zoo embeds a core fragment of OCaml into Rocq and enables machine-checked proofs of fine-grained concurrent data structures. To understand the practical implications of such verification, we examine a data structure based on multi-word compare-and-swap (kCAS)[3] by Rachid Guerraoui, Alex Kogan, Virendra J. Marathe and Igor Zablotchi, §5 originally proposed in practical form by Harris and Fraser [4]. Unlike single-word CAS, kCAS supports atomic updates to multiple memory locations.

Zoo [1] assumes sequential consistency, whereas real-world Multicore OCaml operates under a relaxed memory model. To address this discrepancy, we study Cosmo [6], §6, by Mével, Glen Jourdan, Jacques-Henri and François Pottier, a concurrent separation logic tailored to OCaml's weak memory semantics. Cosmo reconciles high-level reasoning principles with low-level memory behavior, exposing the additional subtleties introduced by relaxed memory.

Finally, we consider Reagents [7], §7, by Aaron Turon, which propose a transactional abstraction for composing fine-grained concurrent interactions. Reagents structure lock-free algorithms around explicit commit boundaries, offering a compositional model that resembles transactional memory while retaining scalability. Through these successive acts, ie. from specification, to automation, to mechanization, to memory-model awareness, and ultimately to compositional concurrency the report explores how modern verification techniques engage with increasingly expressive concurrent programming paradigms.

[*]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*.

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.

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 [9]

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

4.1 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.

4.2 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 non-deterministic with carefully specified guarantees. They also introduce a mechanism for controlling unsharing through generative constructors, enabling sound reasoning.

4.3 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. The project remains actively developed. During the past semester, I have begun contributing to the codebase and engaging with the maintainers, including submitting improvements and participating in ongoing discussions.[1]

[1] https://github.com/clef-men/zoo/pull/2

5 Efficient Multi-word Compare and Swap[3]

5.1 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.

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

5.2.1 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:

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

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.

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

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:

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

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:

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

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.

5.3 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:

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

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:

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

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:

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

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:

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

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

5.4 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.

6 Cosmo: a concurrent separation logic for multicore OCaml [6]

6.1 Motivation

So far, we have established a framework for verifying concurrent programs under sequential consistency (SC). Under SC, the execution of a concurrent program can be understood as an interleaving of thread-local actions over a single, centralized shared memory, which makes reasoning about correctness comparatively straightforward.

However, real-world programming languages, such as OCaml, do not operate under sequential consistency. Instead, they adopt relaxed (or weak) memory models, where program executions are not necessarily representable as simple interleavings of thread actions. As a result, correctness reasoning must account for the semantics of the underlying memory model before addressing higher-level concurrent behavior.

To reason about concurrent programs in such languages, one must first reason at a low level, directly in terms of the memory model and its allowed reorderings and visibility constraints. The terms weak memory model and relaxed memory model are used interchangeably, both referring to the fact that program execution is no longer constrained to behave as if all threads interact with a single, sequentially consistent memory. Instead, memory effects may be delayed, reordered, or observed differently by different threads.

Once a framework for low-level reasoning is in place, it becomes possible to reason modularly about libraries of

$$\begin{aligned}
a &\in \mathsf{Loc}_{\mathsf{NA}} \\
A &\in \mathsf{Loc}_{\mathsf{AT}} \\
t \in \mathsf{Time} &\triangleq \mathbb{Q} \cap [0, \infty) \\
h \in \mathsf{Hist} &\triangleq \mathsf{Time} \xrightarrow{fin} \mathsf{Val} \\
V, W, G \in \mathsf{View} &\triangleq \mathsf{Loc}_{\mathsf{NA}} \xrightarrow{set} \mathsf{Time} \\
\sigma \in \mathsf{Store} &\triangleq (\mathsf{Loc}_{\mathsf{NA}} \xrightarrow{fin} \mathsf{Hist}) \times (\mathsf{Loc}_{\mathsf{AT}} \xrightarrow{fin} (\mathsf{Val} \times \mathsf{View}))
\end{aligned}$$

Figure 1. Semantic objects: locations, time stamps, histories, views, and stores.

concurrent data structures, and subsequently about the high-level programs that rely on them.

Fortunately, the Multicore OCaml memory model is relatively simple when compared to many industrial weak memory models. It distinguishes between two kinds of memory locations: atomic and nonatomic. Atomic locations come with well-defined synchronization guarantees, whereas nonatomic locations must be accessed in a data-race-free manner to ensure meaningful behavior.

Reasoning about atomic locations follows a similar spirit to the approach discussed in earlier work on linearizability-based verification [9]. In particular, when an operation performs a stateful change, it becomes possible to identify a clear correctness condition tied to the update. In contrast, stateless operations require the logic to retain and compose all relevant information to justify correctness.

Cosmo builds precisely such a framework: one that supports low-level reasoning about relaxed memory behaviors while enabling compositional, high-level reasoning about concurrent programs and data structures.

6.2 Proposed Solution

Cosmo makes the underlying relaxed memory model explicit using a small collection of semantic objects: locations, timestamps, histories, views, and stores. Non-atomic locations are associated with histories (h) that record write events as timestamp–value pairs, while atomic locations store both a value and an associated view (V). A view is a mapping from non-atomic locations to timestamps and represents the latest writes that are visible. Each thread carries its own view (W), which records what that thread is currently allowed to observe. The store σ aggregates these components, enabling the logic to precisely describe how writes are created, ordered, and observed, and to distinguish between operations that merely consult visibility and those that propagate it.

For the purposes of this report, to maintain brevity and to pepper my understanding of the paper while building

Mem-NA-Alloc
$$\frac{a \notin \operatorname{dom}\sigma \quad h = \{0 \mapsto v\}}{\sigma; W \xrightarrow{\text{alloc}(a,v)} \sigma[a \mapsto h]; W}$$

Mem-AT-Alloc
$$\frac{A \notin \operatorname{dom}\sigma}{\sigma; W \xrightarrow{\text{alloc}(A,v)} \sigma[A \mapsto (v, W)]; W}$$

Mem-NA-Read
$$\frac{h = \sigma(a) \quad t \in \operatorname{dom} h \quad W(a) \le t \quad v = h(t)}{\sigma; W \xrightarrow{\text{rd}(a,v)} \sigma; W}$$

Mem-AT-Read
$$\frac{\sigma(A) = (v, V)}{\sigma; W \xrightarrow{\text{rd}(A,v)} \sigma; W \cup V}$$

Mem-NA-Write
$$\frac{h = \sigma(a) \quad t \notin \operatorname{dom} h \quad W(a) < t \quad h' = h[t \mapsto v]}{\sigma; W \xrightarrow{\text{wr}(a,v)} \sigma[a \mapsto h']; W[a \mapsto t]}$$

Mem-AT-Write
$$\frac{\sigma(A) = (v, V) \quad V' = W' = W \cup V}{\sigma; W \xrightarrow{\text{wr}(A,v')} \sigma[A \mapsto (v', V')]; W'}$$

Mem-AT-Read-Write
$$\frac{\sigma(A) = (v, V) \quad V' = W' = W \cup V}{\sigma; W \xrightarrow{\text{rdwr}(A,v,v')} \sigma[A \mapsto (v', V')]; W'}$$

Figure 2. Operational semantics for memory actions over nonatomic and atomic locations.

intuition, we adopt a Manhattan-style city-grid analogy. [2] Each memory location is a building whose floors correspond to writes over time. The view (W) is the photograph of the city carried by the active thread: it may be incomplete and merely inspecting a building using this photograph does not update it. The view (V) is a photograph stored inside an atomic building, left behind by an atomic write and retrievable by other threads. Non-atomic reads and writes consult only (W), do not store or propagate photographs, and may therefore observe writes that are not globally latest but are nonetheless visible according to the thread's current snapshot. Atomic operations, in contrast, merge (W) with (V), explicitly propagating visibility. Allowing such stale non-atomic reads is not a weakness of the model; it faithfully captures the fact that, under relaxed memory, threads only know what their current view (W) entitles them to know, and synchronization must be made explicit through atomic operations.

[2] The "Manhattan" grid analogy and a literature that uses *atomics* quite frequently are purely coincidental, despite the historical temptation.

BaseCosmo

The operational semantics of Cosmo describe how a program executes at the machine level: an expression (e) reduces to a new expression (e'), possibly interacting with the memory subsystem through a memory event and possibly spawning new threads. These semantics precisely track how memory locations, histories, timestamps, and views evolve during execution. However, while such a semantics tells us *what can happen*, it does not by itself tell us *how to reason compositionally* about programs, or how to prove that a program is safe or correct.

Cosmo instantiates Iris, a generic framework for building concurrent separation logics. Iris provides the logical infrastructure, assertions, ghost state, invariants, weakest preconditions, and proof rules, while BaseCosmo supplies the Multicore OCaml–specific ingredients.

At this point, the Manhattan city-grid analogy becomes useful again. In the operational model, each memory location is a building whose floors are writes, histories record all floors ever built, and views determine how tall each building appears to a given thread. BaseCosmo's role is to lift this concrete picture into logic. Instead of directly manipulating buildings and photographs, the logic reasons about *ownership* and *knowledge.* Fractional permissions allow ownership of buildings to be split across threads, while a global view invariant ensures that every thread's snapshot (W) remains consistent with the actual city skyline represented by the store. In other words, BaseCosmo ensures that the logical "photographs" carried by proofs faithfully correspond to the operational photographs carried by threads.

To make this precise, BaseCosmo introduces assertions such as points-to assertions and a valid-view assertion. Points-to assertions describe ownership and knowledge of individual memory locations both nonatomic and atomic, while the valid-view assertion captures the global invariant relating thread views to the store. Atomic points-to assertions additionally account for the fact that atomic locations store a view (V), which can later be merged into a thread's view (W), explicitly propagating visibility.

Hoare triples are the interface through which all of this reasoning is exposed. A triple (P, (e, W), Φ) states that if the precondition (P) holds, then executing expression (e) in a thread with view (W) is safe, and if it terminates, it produces a value and an updated view satisfying the postcondition (Φ). Iris supplies the generic machinery for weakest preconditions, framing, and invariants, while BaseCosmo provides the memory-specific axioms that explain how reads, writes, allocations, and atomic operations affect ownership and views.

A Higher Level Logic : Cosmo

While BaseCosmo provides a faithful translation of the Multicore OCaml operational semantics, it remains intentionally low-level. In particular, it exposes details such as histories, timestamps, and per-thread views directly in assertions. This precision is useful for soundness, but it makes reasoning cumbersome: even in data-race-free code, a nonatomic location appears to store an entire history rather than a single value, and every Hoare triple must explicitly mention the current thread's view (W).

Finally, all the preamble data takes place. Its goal is not to change the underlying semantics, but to provide a more convenient interface for reasoning. On top of BaseCosmo, Cosmo offers a higher-level logic in which nonatomic locations can again be reasoned about as if they store a single value, provided the program is data-race free while still remaining sound with respect to relaxed memory.

Back to our analogy - at the BaseCosmo level, reasoning involves explicit references to building histories and photographs: which floors exist, which timestamps label them, and which buildings are visible in a thread's snapshot. Cosmo abstracts over these details. From the perspective of Cosmo, a nonatomic building simply "contains a value," meaning that the most recent write to that building is known to the current thread.

Instead of asserting a BaseCosmo proposition directly, a Cosmo assertion denotes a function from views to BaseCosmo assertions. Subjective assertions, such as "this thread has seen view (V)," depend on the current thread's snapshot and therefore cannot be shared. Objective assertions, on the other hand, are independent of any particular thread's view and can be placed inside invariants. This distinction explains why nonatomic points-to assertions in Cosmo are necessarily subjective: they assert not only that a value exists, but that the current thread is aware of the corresponding write.

Iris continues to play a central role in this construction. It provides the general machinery for weakest preconditions, Hoare triples, framing, and invariants, while Cosmo refines the assertion language to separate subjective, view-dependent knowledge from objective, shareable facts. Hoare triples in Cosmo therefore look familiar: a precondition (P) describes the resources required to execute an expression (e), and the postcondition Φ describes the resources obtained after execution. What changes is that Cosmo carefully controls how view-dependent information flows through these triples, ensuring monotonicity with respect to view growth and preserving soundness under relaxed memory.

6.3 Conclusion

This paper introduced Cosmo, a concurrent separation logic that enables formal reasoning about Multicore OCaml under its relaxed memory model. Unlike traditional program logics that assume sequential consistency, Cosmo is designed to reason directly about the realities of modern multicore execution, including delayed visibility, partial views, and explicit synchronization.

The broader goal of this paper was also to understand how the verification of concurrent programs evolve as we move closer to real systems. While Cosmo significantly advances the state of the art by enabling formal reasoning about Multicore OCaml under a relaxed memory model, it is not the end of the story. As indicated by the authors, several important directions remain open, including support for arrays, richer synchronization primitives such as the Domain API, and full verification of sophisticated lock-free data structures. More fundamentally, reasoning about programs that intentionally exploit data races on nonatomic locations remains an open challenge.

7 Reagents: expressing and composing fine-grained concurrency [7][8]

7.1 Motivation

Amidst a sea of non-blocking progress guarantees, wait-free algorithms offer the strongest promise, every thread completes in a bounded number of steps regardless of interference, but are difficult to design and reason about. At the other end, obstruction-free algorithms guarantee progress only in the absence of contention, making them easier to construct but weaker in practice. Lock-free data structures occupy the practical sweet spot: they ensure system-wide progress (at least one thread completes) while remaining implementable and scalable. Modern libraries such as `java.util.concurrent` provide highly optimized lock-free stacks, queues, and maps, yet these structures are not *composable*. For instance, performing `x = A.pop(); B.push(x);` is not atomic; between the two calls, another thread may observe inconsistent intermediate state. Extending the library with ad hoc methods like `popAndPush` leads to API explosion and violates a key abstraction principle: libraries should expose composable building blocks, not bespoke combinations. In traditional lock-free programming, the developer must explicitly orchestrate CAS loops, retries, and memory interactions, thereby controlling the low-level protocol. Reagents address this gap by introducing a new execution model. A reagent, written `Reagent`$[A, B]$, is a *description* of a concurrent atomic interaction: given input A, it performs coordinated memory updates and synchronization to produce B, while abstracting away CAS retries, transient failures, and commit protocols. By treating atomic operations as first-class composable values, reagents allow developers to declaratively specify interactions while the runtime enforces lock-free coordination. In doing so, they reconcile scalability with composability, avoiding global locks, heavyweight transactional memory, and abstraction-breaking APIs.

```
ch := make(chan int)
go func() {
  ch <- 10 // send
}()
x := <-ch // receive
```

Figure 3. Basic channel creation and rendezvous in Go.

```
select {
case x := <-ch1:
  fmt.Println("Received␣from␣ch1:", x)
case y := <-ch2:
  fmt.Println("Received␣from␣ch2:", y)
default:
  fmt.Println("No␣channel␣ready")
}
```

Figure 4. Selective communication using `select` in Go.

7.2 Proposed Solution

Reagents attempt to reconcile two historically distinct models of concurrency. The first model is the *shared-state paradigm*, in which threads coordinate by reading and writing shared memory locations using atomic primitives such as compare-and-swap (CAS). In this setting, correctness depends on carefully maintaining invariants over mutable state, and progress is achieved through retry loops that repeatedly attempt atomic updates. Lock-free stacks and queues are canonical examples of this approach.

The second model is the *message-passing paradigm*, in which threads do not interact by sharing memory directly but instead exchange values over channels. Synchronization is achieved through structured communication rather than low-level atomic instructions. This approach emphasizes clarity and composability, as communication events themselves become the unit of coordination.

Reagents unify these traditions by treating atomic memory updates as first-class synchronization events. In effect, they allow CAS-based interactions from the shared-state world to be composed in a manner similar to communication events in the message-passing world.
The send operation (Figure 3) `ch <- 10` blocks until another goroutine performs a corresponding receive `x := <-ch`. This rendezvous ensures that communication and synchronization occur simultaneously, without exposing shared-memory races. Go also provides selective communication through the `select` statement:

The `select` statement waits until one of the communication operations is enabled. If multiple cases are ready, one is chosen non-deterministically. If none are ready and a `default` clause is present, execution proceeds with that branch; otherwise, the goroutine blocks. This construct exemplifies *selective communication*, where the runtime determines which interaction can proceed. The same conceptual mechanism appears in Concurrent ML via the `choose` combinator, where synchronization events are treated as first-class values and composed algebraically.

7.3 Algebra of Reagents

Reagents introduce a small but expressive algebra for composing concurrent atomic interactions. At its core are three combinators: sequencing ($r_1 \quad r_2$), parallel conjunction ($r_1 * r_2$), and choice ($r_1 \mid r_2$). Sequencing composes two reagents transactionally, ensuring that the effect of r_2 logically follows r_1 within a single atomic interaction. Parallel conjunction requires that both r_1 and r_2 be enabled and commit together, forming a coordinated atomic update across multiple memory locations. Choice allows the system to attempt multiple interactions and commit whichever becomes enabled first, reminiscent of selective communication. Unlike Software Transactional Memory (STM), where an `atomic` block tracks all reads and writes within its scope and validates them during commit, reagents expose only explicitly declared atomic updates; speculative reads remain invisible and do not participate in global validation. Composition therefore occurs at well-defined commit boundaries rather than at the level of arbitrary memory accesses. For example, composing pop(A) push(B) yields an atomic transfer between two lock-free stacks without introducing a global transaction that tracks every intermediate read. Similarly, $r_1 \mid r_2$ allows a program to proceed with whichever operation becomes feasible first, and $r_1 * r_2$ coordinates multiple atomic updates as a single interaction. In this way, reagents provide algebraic composition of lock-free operations while avoiding the heavyweight bookkeeping characteristic of STM.

7.4 Implementation Overview

Reagents execute via a conceptually two-phase protocol [8]. When a reagent is invoked (via the `!` method), it attempts to *react*. Reaction consists of (1) a *collection phase*, during which the runtime builds up a `Reaction` object containing the atomic updates (e.g., CAS operations), messages, and post-commit actions; and (2) a *commit phase*, during which the collected updates are atomically applied.

A failure during the first phase (i.e., failure to build the desired reaction) is classified as a *permanent failure*. This corresponds to logical impossibility under current conditions (e.g., popping from an empty stack with no alternative branch). Retrying would not help; the reagent must block until external state changes. By contrast, a failure during the commit phase is a *transient failure*: the reaction was valid, but interference from another thread prevented atomic commitment. In this case, the system retries.

7.5 Reagent Syntax and Semantics

The Reagents library provides a distinct syntax for expressing fine-grained concurrency, centered around the composition of atomic transactions. A Reagent $R[A, B]$ is not a simple function from A to B, but a description of an atomic transaction that transforms an input of type A into a result of type B.

The core operation for executing a reagent is the exclamation mark (`!`), which initiates the transaction:

$$\texttt{result = reagent ! input} \tag{3}$$

This invocation triggers the aforementioned two-phase process: *accumulation*, where the transaction's effects (such as Compare-and-Swap operations) are collected into a `Reaction` object, and *commit*, where these effects are atomically applied.

Reagents are composed using combinators that can be thought of as bind operations. The sequencing combinator (») chains reagents together:

$$R_1 \gg R_2 \tag{4}$$

This expression implies a dependency: "Attempt R_1; if successful, use its result as input for R_2. If either fails, the entire transaction aborts."

Internally, this behavior is implemented via the `tryReact` method:

```
def tryReact(a: A, rx: Reaction, offer:
    Offer[B]): Any
```

Here, `rx` represents the accumulated state of the transaction so far—the pending CAS operations and message sends.

7.6 The Necessity of Continuation-Passing Style (CPS)

The sequential composition $R_1 \gg R_2$ presents a fundamental challenge: R_1 cannot simply return a value to R_2 because the transaction is not yet complete. The system must maintain the ability to abort, retry, or block the entire chain based on future conditions. This requirement necessitates *Continuation-Passing Style (CPS)*.

7.6.1 Inverting Control Flow.

In standard direct-style programming, a function returns a result to its caller. In CPS, a function receives the "rest of the computation" (the continuation) as an argument and decides how to proceed. This is often summarized by the Hollywood Principle: "Don't call us, we'll call you."

The `tryReact` method embodies this principle. The `rx` argument effectively serves as the continuation. Instead of computing a final result, a reagent transforms the current continuation:

```
// The tryReact function transforms an input
     and a continuation
// into either a new continuation or a
    failure.
//
// Syntax Mapping:
// 1. × becomes (A, B): In Scala, a product
     type is a tuple/argument list.
// 2. → becomes =>: The function arrow in
    Scala is =>.
// 3. ∨ becomes |: In Scala 3, A | B is a
    Union Type, mapping "Success OR Failure
    ".

type TryReact = (A, CurrentContinuation) =>
    NewContinuation | Failure
```

Figure 5. Conceptual type signature of TryReact

This allows the library to build up the transaction step-by-step without committing any side effects until the entire chain is ready.

7.6.2 Explicit Failure and Retry.

CPS is structurally required to handle the non-deterministic nature of concurrent programming. A reagent does not just succeed or fail; it has three possible outcomes:

1. **Success:** The reagent extends the continuation with its operations and passes control to the next reagent.
2. **Transient Failure (Retry):** Interference occurred (e.g., a CAS failed). The continuation is discarded, and the transaction restarts.
3. **Permanent Failure (Block):** The operation cannot proceed (e.g., waiting on an empty queue). The thread suspends execution.

By using CPS, the Reagents library treats the execution path as a first-class value that can be suspended, discarded, or re-executed. This explicit management of the continuation allows for the composition of complex, lock-free synchronization primitives—such as the kCAS protocol—that would be impossible to express using standard return values.

7.7 Conclusion

Reagents suggest a transactional abstraction for lock-free data structures with explicit commit boundaries. Such structure may significantly simplify reasoning about linearization points and compositional correctness. Developing a formal operational semantics could therefore provide a foundation for automated linearizability verification in the presence of both shared-state and message-passing interactions.

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] Glen Mével, Jacques-Henri Jourdan, and François Pottier. Cosmo: a concurrent separation logic for multicore ocaml. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020.

[7] Aaron Turon. Reagents: expressing and composing fine-grained concurrency. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 157–168, New York, NY, USA, 2012. Association for Computing Machinery.

[8] Aaron Turon. *Understanding and Expressing Scalable Concurrency*. PhD thesis, Northeastern University, 2013.

[9] 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.