Scenario-Based Proofs for Concurrent Objects

CONSTANTIN ENEA, LIX - CNRS - École Polytechnique, France
ERIC KOSKINEN, Stevens Institute of Technology, USA

Concurrent objects form the foundation of many applications that exploit multicore architectures and their importance has lead to informal correctness arguments, as well as formal proof systems. Correctness arguments (as found in the distributed computing literature) give intuitive descriptions of a few canonical executions or "scenarios" often each with only a few threads, yet it remains unknown as to whether these intuitive arguments have a formal grounding and extend to arbitrary interleavings over unboundedly many threads.

We present a novel proof technique for concurrent objects, based around identifying a small set of scenarios (representative, canonical interleavings), formalized as the commutativity quotient of a concurrent object. We next give an expression language for defining abstractions of the quotient in the form of regular or context-free languages that enable simple proofs of linearizability. These quotient expressions organize unbounded interleavings into a form more amenable to reasoning and make explicit the relationship between implementation-level contention/interference and ADT-level transitions.

We evaluate our work on numerous non-trivial concurrent objects from the literature (including the Michael-Scott queue, Elimination stack, SLS reservation queue, RDCSS and Herlihy-Wing queue). We show that quotients capture the diverse features/complexities of these algorithms, can be used even when linearization points are not straight-forward, correspond to original authors' correctness arguments, and provide some new scenario-based arguments. Finally, we show that discovery of some object's quotients reduces to two-thread reasoning and give an implementation that can derive candidate quotients expressions from source code.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; • **Theory of computation** → **Logic and verification**; **Program reasoning**; • **Computing methodologies** → **Concurrent algorithms**.

Additional Key Words and Phrases: verification, linearizability, commutativity quotient, concurrent objects

ACM Reference Format:
Constantin Enea and Eric Koskinen. 2024. Scenario-Based Proofs for Concurrent Objects. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 140 (April 2024), 30 pages. https://doi.org/10.1145/3649857

1 INTRODUCTION

Efficient multithreaded programs typically rely on optimized implementations of common abstract data types (ADTs) like stacks, queues, and sets, whose operations execute in parallel to maximize efficiency. Synchronization between operations must be minimized to increase throughput [Herlihy and Shavit 2008]. Yet this minimal amount of synchronization must also be adequate to ensure that operations behave as if they were executed atomically, so that client programs can rely on their (sequential) ADT specification; this de-facto correctness criterion is known as *linearizability* [Herlihy and Wing 1990]. These opposing requirements, along with the general challenge in reasoning about interleavings, make concurrent data structures a ripe source of insidious programming errors.

Algorithm designers (*e.g.,* researchers defining new concurrent objects) argue about correctness by considering some number of "scenarios", *i.e.,* interesting ways of interleaving steps of different

Authors' addresses: Constantin Enea, LIX - CNRS - École Polytechnique, Paris, France, cenea@lix.polytechnique.fr; Eric Koskinen, Stevens Institute of Technology, Hoboken, USA, eric.koskinen@stevens.edu.

operations, and showing for instance, that each one satisfies some suitable invariant (which is not necessarily inductive). For example, a scenario of the Michael and Scott [1996] queue is described as: many threads concurrently reading, one enqueuer thread taking a specific read path finding a tail pointer to be outdated, and then succeeding a compare-and-swap (CAS) operation, causing others to fail their compare-and-swap (paraphrasing from Herlihy and Shavit [2008]). Such scenario descriptions are powerful because they describe unboundedly many threads and often generalize to cover many executions that are equivalent due to commutative re-orderings. Consequentially, informal correctness arguments need only consider a few representative scenarios. Furthermore, another critical benefit of scenario-based reasoning is that scenarios are more readily explainable to software developers, who need not have a background in formal logic.

Despite the intuitive benefit of these operational, scenario-based proofs—which continue to be widely used in the concurrent algorithms literature—it remains unknown as to whether they have a formal grounding. This has lead to cases where objects thought to be linearizable [Detlefs et al. 2000] where later determined to contain bugs in unconsidered scenarios [Doherty et al. 2004].

1.1 Formalizing Scenarios with Quotients

In this paper, we show that operational, scenario-based correctness arguments can be formally grounded. To that end, we propose a new proof methodology that is based on formal arguments while keeping the intuition of scenario-based reasoning. This methodology relies on a reduction to reasoning about a subset of *representative* interleavings (i.e. a formal version of informal scenarios), which cover the whole space of interleavings modulo repeatedly swapping adjacent commutative steps. The latter corresponds to the standard *equivalence up to commutativity* between the executions of an object (e.g., Mazurkiewicz traces [Mazurkiewicz 1986]).

Reductions based on commutativity arguments have been formalized in previous work, *e.g.,* Lipton's reduction theory [Lipton 1975], QED [Elmas et al. 2009], CIVL [Hawblitzel et al. 2015], and they generally focus on identifying *atomic sections*, *i.e.,* sequences of statements in a single thread that can be assumed to execute without interruption (without sacrificing completeness). Relying on atomic sections for reducing the space of interleavings has its limitations, especially in the context of concurrent objects. These objects rely on intricate algorithms where almost every step is an access to the shared memory that does not commute with respect to other steps.

Our reduction argument reasons about a *quotient* of the set of object executions, which is a subset of executions that contains a representative from each equivalence class. In general, an execution of an object interleaves an unbounded number of invocations to the object's methods, each from a different thread[1]. These executions can be seen as a word over an infinite alphabet, each symbol of the alphabet representing a statement in the code and the thread executing that statement[2]. We show that when abstracting away thread ids from executions, carefully chosen quotients become *regular or context-free languages*. This is not true for any quotient since representatives of equivalence classes can be chosen in an adversarial manner to make the language more complex.

The principal benefit of quotients is that reasoning about correctness can be done by considering only a few representative execution interleavings, yet those conclusions generalize to all executions. For some kinds of concurrent object implementations (defined later), deriving representative traces can be reduced via induction to two-thread reasoning.

Proofs with program logics. Our work is inspired by the success of many prior works on proofs for concurrent objects based on program logics such as Owicki and Gries [1976], Rely/Guarantee [Jones 1983], Concurrent separation logic [Brookes 2007; O'Hearn 2007], RGSep [Vafeiadis and Parkinson

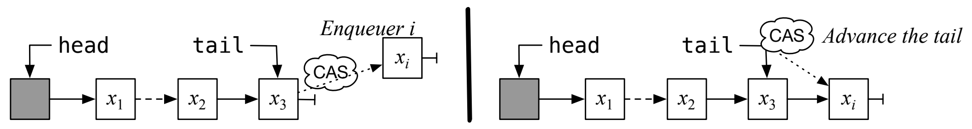
[1]Typically, it can be assumed w.l.o.g. that each thread performs a single invocation in an execution.
[2]Such a sequence will be called a *trace* in the formalization we give later in the paper.

2007], Deny-Guarantee [Dodds et al. 2009], Views [Dinsdale-Young et al. 2013], Iris [Jung et al. 2018, 2015] and interactive proof tools for such logics.

The goal of this paper is orthogonal and focuses on finding a formal grounding for the operational, scenario-based correctness arguments present in the algorithms literature. To this end, our methodology is based on taking representative interleaved traces upfront and using commutativity-based equivalence classes for modularity/generalization rather than exploiting the program structure and invariants for modularity/generalization. Achieving this alternative reasoning strategy nonetheless requires careful formalization of what is meant by "representative traces", as well as how those classes of traces can be expressed abstractly, which we outline below. Our results show that (i) scenario-based reasoning can be done formally through quotients, (ii) quotients can be given for a variety of concurrent objects with subtle differences including non-fixed linearization points, (iii) quotients improve the correctness arguments from the literature, and (iv) for some cases, quotients—which represent interleavings of unboundedly many threads—can be automatically discovered through a reduction to two-thread reasoning.

1.2 Example: Scenario-Based Proofs of the Michael-Scott Queue

For the sake of concreteness, we now show how quotients make concurrent reasoning simpler, using the canonical Michael-Scott Queue (MSQ) as an example. Ultimately the theory and algorithms in this paper lead to an implementation that is able to automatically derive the representation discussed below, from the object's source code. The MSQ is implemented as a linked-list, with head and tail pointers and a sentinel head node, as depicted to the left below.



An enqueue (`enq`) operation, such as *Enqueuer i* in the diagram above, repeatedly attempts to enqueue a new element by using an atomic compare-and-swap (CAS) operation on the `tail` element's `next` pointer, replacing null with the address of the new node (x_i in the diagram above). It is possible that this CAS operation will fail due to a concurrent enqueuer (of which there can be unboundedly many). Nonetheless, due to the CAS, one enqueuer will succeed. At this point, although the element is linked, it is not logically in the queue because the `tail` pointer is lagging. The enqueuer will thus perform a second CAS operation, as shown on the digram above to the right, to advance `tail` to point to x_i. To ensure progress, concurrent enqueuers will also check to see if the `tail` lags and, if so, attempt to advance the `tail` before they attempt to enqueue their elements (i.e. helping). A dequeue (`deq`) operation repeatedly attempts to advance the head pointer to make x_1 the new sentinel node, but also has to check that the queue is non-empty and that other threads have not recently dequeued. (To achieve all of these cases, `deq` must begin by reading the `head` pointer, the `tail` pointer and `head`'s `next` pointer and validating to see which case applies.)

To verify the correctness of objects like the MSQ, one has to consider all of the ways in which concurrent invocations of unboundedly many methods could interleave. One strategy to tackle this problem has been through the aforementioned program logics such as rely-guarantee where, roughly, one defines state-based invariants and then shows they are preserved and threads don't interfere with other threads' actions. Nevertheless, the correctness arguments laid out by algorithm designers (*e.g.*, in the distributed computing community) typically are organized in a more operational manner and instead focus on discussing various "scenarios." Consider the following excerpt from The Art of Multiprocessor Programming [Herlihy and Shavit 2008] regarding the MSQ:

An enqueuer creates a new node, reads tail, and finds the node that appears to be last. To verify that node is indeed last, it checks whether that node has a successor. If so, the thread attempts to append the new node with CAS. (A CAS is required because other threads may be trying the same thing.) [Assume that] the CAS succeeds.

Such sentences describe scenarios that involve unboundedly many threads executing some portion of their programs. They are chosen to highlight tricky situations and describe why those situations are still acceptable. The above example can be thought of as the sequence:

(1) Unboundedly many threads are reading the data structure.
(2) There is a distinguished thread, let's call τ_{enq}.
(3) τ_{enq} reads the `tail` and the `tail`'s `next` pointer.
(4) τ_{enq} finds that `tail`'s `next` is null.
(5) τ_{enq}atomically updates `tail`'s `next` to point to its new node.
(6) The other (unboundedly many) threads fail their CASes on `tail`'s `next` and restart.

This scenario has a particular shape about it: unboundedly many threads read, then a single thread performs a write, then the remaining threads react to that write. This is a common setup in many non-blocking concurrent algorithms and a useful pattern (although, in general, we will describe scenarios beyond those of this shape). One might think of it as a regular expression denoted $r_{\mathtt{next}}$:

$$r_{\mathtt{next}} \equiv (\tau \in T : read + \tau_{enq} : read)^* \cdot (\tau_{enq} : \mathtt{cas/succeed}) \cdot (\tau \in T : restart)^*$$

where T is the (unbounded) set of all threads excluding τ_{enq}. Above $r_{\mathtt{next}}$ expresses that some unboundedly many threads from set T (including τ_{enq}) perform only *read*-path actions, then τ_{enq} succeeds its `cas`, then those unboundedly many threads restart. This expression is more powerful than it may first appear. There are a few important considerations:

- *Conciseness.* The entirety of MSQ's concurrent execution behaviors can be represented with *this and only two other* similarly concise representative interleavings, along with four even simpler read-only interleavings. Expressions $r_{\mathtt{tail}}$ and $r_{\mathtt{head}}$ are similarly defined and represent advancing the tail pointer and the head pointer (due to a dequeuer), respectively.
- *Unbounded.* With these concise descriptions, the interleavings between an unbounded number of enqueuers and dequeuers can be seen as an unbounded alternation $(r_{\mathtt{next}} + r_{\mathtt{tail}} + r_{\mathtt{head}})^*$. (Below we will further refine this approximation with stateful automata.)

This starred-union description does not include all possible ways of interleaving steps of enqueuers, *e.g.,* it does not include interleavings where a thread restarts after two successful CASs since it last read the shared memory. It includes just a subset of representatives that we call a quotient, which is succinct enough to correspond to the designer's intuition and large enough to cover the whole space of interleavings modulo repeatedly swapping adjacent commutative steps (*i.e.,* the standard equivalence up to commutativity between executions known as Mazurkiewicz traces [Mazurkiewicz 1986]). For instance, an interleaving where a thread restarts after two successful CASs (since it last read the shared memory) is equivalent to one where the restart step is reordered to the left to occur immediately after the first CAS. This is because the restarting condition is fulfilled after this first CAS as well and the restart step does not perform any writes.

The MSQ falls into a special class of objects for which quotients can be expressed in this inductive way, as a sequence of what we call "layers" (above $r_{\mathtt{next}}$, $r_{\mathtt{tail}}$ and $r_{\mathtt{head}}$ are layers) wherein only a single shared memory *write* action occurs per layer, and all other actions are thread-local/read-only (perhaps restarting due to a failed CAS). Consequently, it is possible via induction to reduce reasoning to a collection of two-threaded arguments (one writer, one reader). While quotients and their abstractions are a much broader class, layers are nonetheless an important subclass since they apply to many lock-free implementations and can be automated, as discussed below.

1.3 Challenges and Contributions

We now identify several challenges toward enabling scenario-based reasoning and discuss how we address them in this paper.

1. Concurrent Object Quotients. *How can scenario-based reasoning be done formally?* (Sec. 3) We show that scenario-based reasoning can be made formal through a methodology wherein reasoning about all executions of a concurrent object is reduced to reasoning only about a smaller set of representative interleavings. At the technical core is the definition of an object's execution *quotient* which collapses executions that are equivalent up to swapping commutative adjacent actions. A quotient is parameterized by this equivalence relation and has both a minimality constraint (no two executions are equivalent) and a completeness constraint (all executions are equivalent to some execution in the quotient). We prove that linearizability of the quotient is sufficient to show linearizability of the object. The upshot is that concurrent object correctness is now accomplished via reasoning about a collection of scenarios (as in typical informal proofs).

2. Expressing Quotients. *How can a quotient set be described?* (Sec. 4) A next question is how to *finitely express* a quotient, which can have unboundedly many interleavings. In Sec. 3, we introduce a *quotient expression language* that permits a mixture of regular expressions (*e.g.,* Kleene-star iterations of subexpressions) and context-free grammars (*e.g.,* unbounded but balanced subexpressions). We then give an interpretation/semantics for these expressions that maintains the *minimality* condition: there will only be one interleaving (with threads organized in a canonical order) for every unboundedly many unrolling. The MSQ expression $(r_{\texttt{next}} + r_{\texttt{tail}} + r_{\texttt{head}})^*$ above provides an intuition for the quotient expression for the MSQ. (Technically, the *read* actions are paths and the $*$-iterations within the r_{x} subexpressions are replaced with a context-free form of iteration.)

As we will show later, quotients and their abstractions are *expressive* and can capture canonical concurrent objects as well as more complicated ones such as the Herlihy and Wing [1990] queue and the elimination stack of Hendler et al. [2004], each having different kinds of non-fixed linearization points. These are notoriously hard cases for today's proof methodologies. We note that, while the idea of reasoning about execution quotients is generic, identifying precise limits for the applicability of the particular class of quotients expressions is hard in general. This is similar to using abstract domains in the context of static analysis: it is hard to determine precisely the class of programs for which interval or polyhedra abstractions are effective.

3. Layer Quotient Expressions and Automata. (Sec. 5) *In addition to broad expressivity, are there classes of objects whose quotients have a simpler structure?* To increase accessibility and automation, we next describe certain kinds of quotient expressions for which reasoning can actually be reduced, via induction, to two-thread reasoning. Specifically, for objects whose implementation can be written as a collection of (possibly restarting) read-only/local paths and paths that have only a single atomic read-write, we define *layer quotients* to more conveniently and inductively capture the quotient. Although this does not apply to all objects, it does apply to canonical examples such as the MSQ, Treiber's Stack, and even the Scherer III et al. [2006] synchronous reservation queue. For these objects, executions can be decompiled into a sequence of *layers*, each described by context-free quotient expressions of the form $(a_1 + b_1 + \ldots)^n \cdot w \cdot (a_2 + b_2 + \ldots)^n$ where $a_1 \cdot a_2$ is a read-only path through the method implementation (possibly restarting), and w is a path with a successful atomic read-write. The exponents in both expressions indicate the unbounded replication of local paths (n is not fixed; it ensures prefix/suffix balancing). Then an overall quotient expression can be made from regular compositions of these context-free layers, leading to an inductive argument. Furthermore, each layer can be discovered with two-thread reasoning: considering how each write, treated atomically, impacts each other read-only/local path.

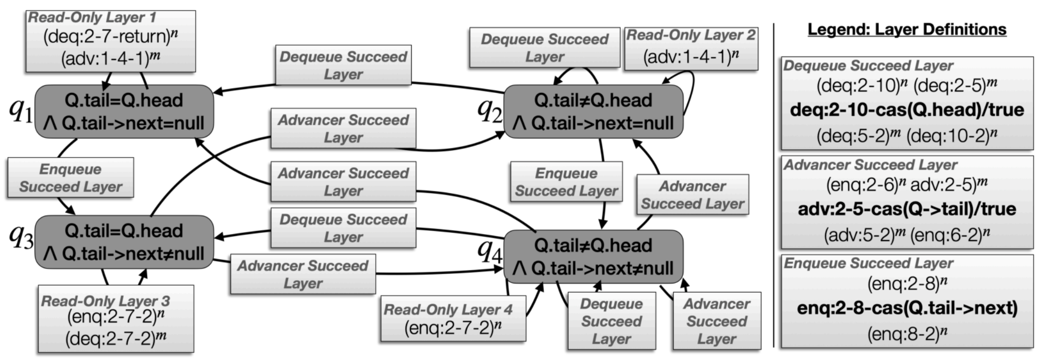


Fig. 1. Layer automaton for the Michael/Scott Queue.

We describe how layer expressions can be conveniently represented as finite-state *automata* (and further below also used for automation). The layer automaton for the Michael-Scott Queue is shown in Fig. 1. We will discuss it in detail in Sec. 6.1 but, roughly, the states track whether the queue is empty and whether the tail is lagging. The layer-labeled edges define the local/read-only (unbold) control-flow paths and how they are impacted by the write path (bold). There are also *read-only* layers, which we will describe later.

4. Evaluation: Verifying Concurrent Objects. (Sec. 6) We consider a broad range of concurrent objects including Treiber's stack [Treiber 1986], the Michael and Scott [1996] queue, the Scherer III et al. [2006] synchronous reservation queue, the Herlihy and Wing [1990] queue, the Hendler et al. [2004] elimination stack, and the Restricted Double-Compare Single-Swap (RDCSS) [Harris et al. 2002]. Each object has its own subtleties, including complications like multiple CAS steps and non-fixed linearization points. For each object we (i) show that its behavior and linearizability can be captured through a quotient and (ii) revisit the object's authors' correctness arguments. We find that quotients capture those intuitive scenarios and make scenarios explicit and comprehensive.

5. Generating Candidate Quotient Expressions. (Sec. 7) Automating quotient-based proofs of concurrent objects is a rather large question (perhaps warranting new forms of induction) which we mostly leave to future work. Nonetheless, we present an algorithm and prototype implementation Cion for generating candidate quotient expressions, directly from a concurrent object's source code. We manually confirmed that these expressions are sound abstractions of those objects' quotients. We applied Cion to layer-compatible objects such as Treiber's Stack and the Michael/Scott Queue, finding that candidate layer expressions can be discovered in a few minutes.

For lack of space, some detail has been omitted and is available in the extended version [Enea et al. 2023] of this paper. Our implementation Cion *is available on GitHub*[3]*, along with benchmark sources.*

2 PRELIMINARIES

Running example: A simple concurrent counter. Fig. 2 lists a concurrent counter with methods for incrementing and decrementing. Both methods of the counter return the value of the counter before modifying it, and the counter is decremented only if it is strictly positive.

Each method consists of a retry-loop that reads the shared variable `ctr` representing the counter and tries to update it using a Compare-And-Swap (CAS). A CAS atomically tests whether `ctr` equals the second argument and if this is the case, then it assigns the value specified by the third

[3]https://github.com/quotientprovers/cion

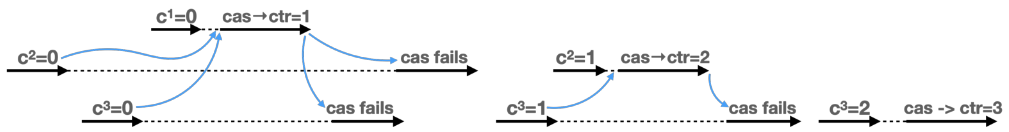


Fig. 3. The steps of an execution with three increment-only threads whose actions are aligned horizontally. For readability, we rename the local variable c in thread i to c^i. The curved blue arrows depict data-flow dependencies between reads/writes of ctr.

argument. If the test fails, then the CAS has no effect. The return value of CAS represents the truth value of the equality test. If the CAS is unsuccessful, *i.e.,* it returns *false*, then the method retries the same steps in another iteration.

```
 1 int increment() {
 2   while (true) {
 3     int c = ctr;
 4     if (CAS(ctr,c,c+1))
 5       return c;
 6   }
 7 }
 8 int decrement() {
 9   while (true) {
10     int c = ctr;
11     if ( c == 0 )
12       return 0;
13     if (CAS(ctr,c,c-1))
14       return c;
15   }
16 }
```

Fig. 2. A concurrent counter.

The executions of the concurrent counter are interleavings of an arbitrary number of increment or decrement invocations from an arbitrary number of threads. Each invocation executes a number of retry-loop iterations until reaching the `return`. An iteration corresponds to a control-flow path that starts at the beginning of the loop and ends with a `return` or goes back to the beginning. For instance, the increment method consists of two possible iterations:

(1) `c = ctr; CAS(ctr, c, c+1); return c`, and
(2) `c = ctr; assume(ctr != c)`.

Iteration #1 is called *successful* because it contains a successful CAS, and the unsuccessful CAS in the iteration #2 is written as an `assume` that blocks if the condition is not satisfied.

An invocation can execute more iterations if ctr is modified by another thread in between reading it at line 3 or 10 and executing the CAS at line 4 or 13, respectively. Fig. 3 pictures an execution with 3 increments that execute between 1 and 3 retry-loop iterations. The first iteration of threads 2 and 3 contains unsuccessful CASs because thread 1 executed a successful CAS and modified ctr, and these invocations must retry, execute more iterations. Note that there are unboundedly many such executions and, even with bounded threads, exponentially many interleavings.

Concurrent Object Syntax. We model concurrent objects using Kleene Algebra with Tests [Kozen 1997] (KAT). Intuitively, a KAT represents the code of an object method using regular expressions over symbols that represent conditionals (tests) or statements (actions).

Definition 2.1. [Kleene Algebra with Tests] A KAT $\mathcal{K}$ is a two-sorted structure $(\Sigma, \mathcal{B}, +, \cdot, *, \bar{\ }, 0, 1)$, where $(\Sigma, +, \cdot, *, 0, 1)$ is a Kleene algebra, $(\mathcal{B}, +, \cdot, \bar{\ }, 0, 1)$ is a Boolean algebra, and the latter is a subalgebra of the former. There are two sets of symbols: A for primitive actions, and B for primitive tests. The grammar of boolean test expressions is $BExp ::= b \in \mathsf{B} \mid b \cdot b \mid b + b \mid \bar{b} \mid 0 \mid 1$, and the grammar of KAT expressions is $KExp ::= a \in \mathsf{A} \mid b \in BExp \mid k \cdot k \mid k + k \mid k^* \mid 0 \mid 1$. For $k_1, k_2 \in \mathcal{K}$, we write $k_1 \leq k_2$ if $k_1 + k_2 = k_2$.

The primitive actions and tests used in examples in this paper will be along the lines of A = {x := y, x.f := y, ...} and B = {x = y, x.f = y, x = null, x.f = null ...}.

*A*tomic read-write (ARW). We conservatively extend KAT with a syntactic notation ⟪b·a⟫, used to indicate a condition b and action a, between which no other actions can be interleaved. Apart from restricting interleaving (defined below), this does not impact the semantics so it can be represented with two special symbols "⟪" and "⟫" whose semantics are the identity relation. For example a compare-and-swap `cas(x,v,v')` can be represented as $(\langle\!\langle \texttt{[x=v]} \cdot \texttt{x:=v'} \rangle\!\rangle \cdot k) + (\overline{\texttt{[x=v]} \cdot k'})$, where $[x = v]$ is a primitive test and the assignment is a primitive action. Overline indicates negation, as in KAT notation. k is the code to be executed when `cas` succeeds and k' when it fails.

Methods of a concurrent object. We define a method signature $m(\vec{x})/\vec{v}$ with a vector of arguments $\vec{x}$ and return values $\vec{v}$ (often a singleton v). For a vector $\vec{x}$, x_i denotes its i-th component. An *implementation* of a method m is a KAT expression k_m, whose actions may refer to argument values, *e.g.,* $\mathsf{x} := \mathit{args}_i$. A *concurrent object* O is a set of methods $O = \{m_1(\vec{x}_1)/\vec{v}_1 : k_{m_1}, \ldots\}$, associating signatures with implementations. The set of method names in an object O is denoted by $\mathit{Meth}(O)$.

Example 2.2. The counter from Sec. 2 is formalized as $O_{ctr} = \{\texttt{inc()}/v : k_{inc}, \texttt{dec()}/u : k_{dec}\}$

$$k_{inc} = (\texttt{c:=ctr} \cdot \Big((\langle \texttt{[c=ctr]} \cdot \texttt{ctr:=c+1} \rangle \cdot \mathsf{ret}(\mathsf{c})) + (\overline{\texttt{[c=ctr]}})\Big))^*$$

$$k_{dec} = (\texttt{c:=ctr} \cdot \Big((\texttt{[c=0]} \cdot \mathsf{ret}(0)) + (\overline{\texttt{[c=0]}} \cdot \langle \texttt{[c=ctr]} \cdot \texttt{ctr:=c-1} \rangle \cdot \mathsf{ret}(\mathsf{c})) + (\overline{\texttt{[c=ctr]}})\Big))^*$$

The outer * in k_{inc} corresponds to the `while (true)` loop in the method `increment` while the inner + corresponds to the two branches of the conditional. The KAT expression k_{inc} represents every control-flow path of `increment` which goes a number of times through the assignment `c:=ctr` and the "`false`" branch of the conditional before succeeding the atomic read-write and returning (other sequences represented by this regular expression, *e.g.,* , iterating multiple times through the atomic read-write and return will be excluded when defining the semantics).

Concurrent Object Semantics. A full semantics for these concurrent objects is given in the extended version [Enea et al. 2023]. In brief, the semantics involves local states $\sigma_l \in \Sigma_{lo}$, shared states $\sigma_g \in \Sigma_{gl}$, and nondeterministic thread-local transition relation $\sigma_l, \sigma_g, k \downarrow_\ell \sigma_l', \sigma_g', k'$, which optionally involve label ℓ (k and k' are KAT expressions representing code to be executed). These ***labels*** are taken from the set of possible labels $\mathcal{L} \subseteq \mathsf{A} \cup \mathsf{B} \cup \mathsf{call}\ m(\vec{v}) \cup \mathsf{ret}(\vec{v}) \cup \langle b \cdot a \rangle$ which includes primitive actions, primitive tests, call actions, return actions or ARWs. (We here write call $m(\vec{v})$ with free variables to refer to the set of all call actions and similar for returns and ARWs.) Next, a configuration $C = (\sigma_g, T)$ where $T : \mathcal{T} \rightharpoonup (\Sigma_{lo} \times (\mathcal{K} \cup \{\bot\}))$ comprises a shared state $\sigma_g \in \Sigma_{gl}$ and a mapping for each active thread to its local state and current code. We use $\mathcal{T}$ to denote the set of thread ids, which is equipped with a total order $<$. Configurations of an object transition according to the relation $\Rightarrow: C \times (\mathcal{T} \times \mathcal{L}) \times C$, labeled with a thread id and a label.

An object O is acted on by a finite ***environment*** $\mathcal{E} : \mathcal{T} \to O \times \vec{\mathit{Val}}$, specifying which threads invoke which methods, with which argument values. Val denotes a set of values and $\vec{\mathit{Val}}$ denotes the set of tuples of values. We assume that object methods can not access thread identifiers (which is true for concurrent objects defined in the literature) and therefore, each invocation is assumed to be executed by a different thread. An ***execution*** of O in the environment $\mathcal{E}$ is a sequence of labeled transitions between configurations $C_0 \Rightarrow \ldots \Rightarrow C_n$ that starts in the initial configuration C_0 w.r.t. $\mathcal{E}$ and ends in configuration C_n. A configuration $C_f = ({\sigma_g}^f, T^f)$ is ***final*** iff $T^f(t) = (\sigma_l, \bot)$, for some σ_l, for all $t \in \mathsf{dom}(T^f)$. An execution is ***completed*** if it ends in a final configuration. $[\![O \otimes \mathcal{E}]\!]$ denotes the set of completed executions of O in the environment $\mathcal{E}$. A ***trace*** $\tau \in \mathit{Traces}$ is a sequence of $\mathcal{T} \times \mathcal{L}$ pairs, *i.e.,* thread-indexed labels $t_0 : \ell_0, \ldots, t_n : \ell_n$. A trace of an execution ρ denoted τ_ρ is a projection of the thread-indexed labels out of the transitions in the execution.

The ***semantics*** $[\![O]\!]$ of a concurrent object O is defined as the set of traces under all possible environments (*i.e.,* for any number of threads invoking any methods with any inputs). Formally, $[\![O]\!] = \{\tau_\rho \mid \rho \in [\![O \otimes \mathcal{E}]\!]$, for some environment $\mathcal{E}\}$.

Linearizability For an object O, an ***operation*** symbol (or operation for short) $o = m(\vec{u})/\vec{w}$ represents an invocation of a method $m \in \mathit{Meth}(O)$ with signature $m(\vec{x})/\vec{v}$, where $\vec{u}$ is a vector of values for the corresponding arguments $\vec{x}$, and $\vec{w}$ is a vector of values for the corresponding returns

$\vec{v}$. A ***sequential specification*** S for an object O is a set of sequences over operation symbols. For instance, the sequential specification for the counter object includes sequences of increments and decrements corresponding to executions where each invocation executes in isolation, *e.g.*, $\texttt{inc()/0} \cdot \texttt{inc()/1} \cdot \texttt{inc()/2}$ or $\texttt{inc()/0} \cdot \texttt{dec()/1} \cdot \texttt{dec()/0}$.

A trace τ of an object O is ***linearizable*** w.r.t. a specification S if there exists a (linearization-point) mapping $lp(\tau) : \mathcal{T} \rightarrow \mathbb{N}$ where the label at position (index) $lp(\tau)$ in τ is considered to be the so-called *linearization point* of t's invocation, and must satisfy the following:

(1) the position $lp(\tau)$ is after t's invocation label and before t's return,
(2) the (linearization) sequence $lin(\tau, lp)$ of operation symbols $m(\vec{u})/\vec{w}$, where the i-th symbol represents the invocation of the i-th thread t w.r.t. the positions $lp(\tau, t)$, belongs to S.

For example, Fig. 3 pictures a trace which is linearizable w.r.t. the counter specification described above because there exists a linearization-point mapping lp which associates each thread i with the position of the i-th successful CAS. The linearization $\texttt{inc()/0} \cdot \texttt{inc()/1} \cdot \texttt{inc()/2}$ induced by this mapping is admitted by the specification.

For simplicity, we omit invocation labels from traces and consider the first instruction in an invocation to play the same role. Object O is ***linearizable*** wrt a spec. S if all traces in $[\![O]\!]$ are linearizable wrt S.

3 OBJECT QUOTIENTS

To formalize scenarios, we introduce the concept of a *quotient* of an object which is a subset of its traces that represents every other trace modulo reordering of commutative steps or renaming thread ids. For an expert reader, the quotient is a partial order reduction [Godefroid 1996] composed with a symmetry reduction [Clarke et al. 1998] of its set of traces. In general, an object may admit multiple quotients, but as we show later, there exist quotients which can be finitely-represented using regular expressions or extensions thereof. We interpret scenarios as components (sub-expressions) of these finite representations.

Two executions ρ_1 and ρ_2 are ***equivalent up to commutativity***, denoted as $\rho_1 \equiv \rho_2$, if ρ_2 can be obtained from ρ_1 (or vice-versa) by repeatedly swapping adjacent commutative steps. An execution ρ_2 is obtained from ρ_1 through one swap of adjacent commutative steps, denoted as $\rho_1 \equiv_1 \rho_2$, if

$$\rho_1 = C_0^{\mathcal{E}} \cdots C_i \xrightarrow{(t:\ell)} C_{i+1} \xrightarrow{(t':\ell')} C_{i+2} \cdots C_n, \text{ and } \rho_2 = C_0^{\mathcal{E}} \cdots C_i \xrightarrow{(t':\ell')} C'_{i+1} \xrightarrow{(t:\ell)} C_{i+2} \cdots C_n$$

(ρ_2 is obtained from ρ_1 by re-ordering the steps labeled by $t : \ell$ and $t' : \ell'$). When there exist executions ρ_1 and ρ_2 as above, we say that the re-ordered labels ℓ and ℓ' are ***possibly commutative***.

Definition 3.1. The equivalence relation $\equiv \subseteq \mathcal{E} \times \mathcal{E}$ between executions is the least reflexive-transitive relation that includes $\equiv_1$.

The relation $\equiv$ is extended to traces as expected: $\tau_1 \equiv \tau_2$ if τ_1 and τ_2 are traces of executions ρ_1 and ρ_2, respectively, and $\rho_1 \equiv \rho_2$.

For example, the Counter executions below are equivalent up to commutativity (related by $\equiv_1$):

$$\rho = C_0 \cdots C_1 \xrightarrow{(t:\overline{[\texttt{c}_t\texttt{=ctr}]})} C_2 \xrightarrow{(t':\texttt{c}_{t'}\texttt{:=ctr})} C_3 \cdots \text{ and } \rho' = C_0 \cdots C_1 \xrightarrow{(t':\texttt{c}_{t'}\texttt{:=ctr})} C'_2 \xrightarrow{(t:\overline{[\texttt{c}_t\texttt{=ctr}]})} C_3 \cdots$$

assuming that $\texttt{ctr} > 0$ at configuration C_1 (recall that $\overline{[\texttt{c}_t\texttt{=ctr}]}$ represents an unsuccessful CAS).

Definition 3.2. Two traces τ_1 and τ_2 are *equivalent up to thread renaming*, denoted as $\tau_1 \simeq \tau_2$, if there is a bijection α between thread ids in τ_1 and τ_2, resp., s.t. τ_2 is the trace obtained from τ_1 by replacing every thread id label t with $\alpha(t)$.

For example, $C_0 \overset{(t:a)}{\rightsquigarrow} C_1 \overset{(t':b)}{\rightsquigarrow} C_2$ and $C_0 \overset{(t':a)}{\rightsquigarrow} C_1 \overset{(t:b)}{\rightsquigarrow} C_2$ are equivalent up to thread renaming.

We define a quotient of an object as a subset of its traces that is *complete* in the sense that it represents every other trace up to commutative reorderings or thread renaming, and that is *optimal* in that sense that it does not contain two traces that are equivalent up to commutativity. Optimality does *not* include equivalence up to thread renaming (symmetry reduction) because the finite representations we define later abstract away thread ids.

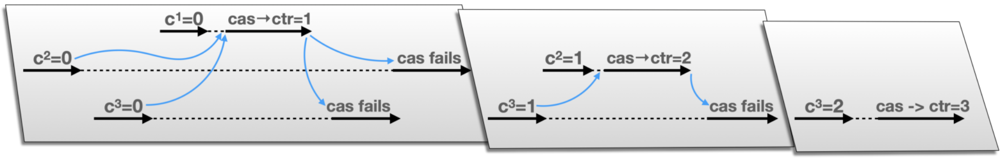
Definition 3.3 (Quotient). A *quotient* of object O is a set of traces $\langle\!\lfloor O \rfloor\!\rangle \subseteq [\![O]\!]$ such that:

- $\forall \tau \in [\![O]\!].\ \exists \tau', \tau''.\tau \simeq \tau' \wedge \tau' \equiv \tau'' \wedge \tau'' \in \langle\!\lfloor O \rfloor\!\rangle$ (completeness), and
- $\forall \tau, \tau' \in \langle\!\lfloor O \rfloor\!\rangle.\ \tau \not\equiv \tau'$ (optimality)

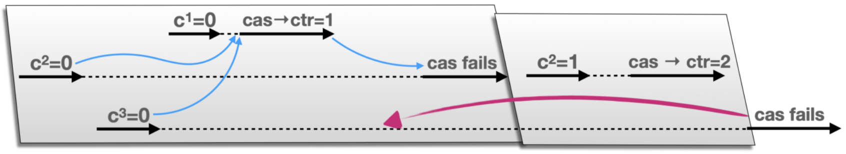
Note that an object admits multiple quotients since representatives of equivalence classes w.r.t. $\equiv$ can be chosen arbitrarily.

For a quotient $\langle\!\lfloor O \rfloor\!\rangle$, a set Swaps of pairs of possibly-commutative labels (in $\mathcal{L} \times \mathcal{L}$) is called $\langle\!\lfloor O \rfloor\!\rangle$***-sufficient*** if all the swaps needed to establish $\tau' \equiv \tau''$ in Def. 3.3 are between pairs of labels in Swaps.

Example 3.4 (Quotient and representative/canonical traces for the Counter). The trace of three increment-only threads from Fig. 3 represents many other traces of the Counter modulo commutative reorderings or thread renaming. It can be thought of as a sequence of three canonical phases, depicted with stacked parallelograms as follows:



Each phase above groups together the retry-loop iterations that interact with each other: a single successful CAS instruction causes the other attempts to fail. For instance, it represents another trace where the first "cas fails" step occurs after the second successful CAS:



This "late" CAS failure would also fail if moved to the left as shown above. Similarly, it also represents traces where the action $c^2 = 0$ is swapped with $c^3 = 0$ and even $c^1 = 0$, or traces where thread ids change from 1, 2, 3 to 4, 5, 6 for instance.

One can define a quotient $\langle\!\lfloor O_{ctr} \rfloor\!\rangle$ of Counter which includes representative traces of this form. The representative traces only differ in the number of incrementers/decrementers and the order in which they succeed their CASs. $\langle\!\lfloor O_{ctr} \rfloor\!\rangle$ will contain similar canonical traces for, say, an environment with 4 incrementers, 2 decrementers acting in the sequence *incr*; *decr*; *decr*; *incr*; *incr*; *incr* (wherein the second *decr* does nothing). See Example 4.3 for a more precise description.

Preserving Linearizability Through Commutative Reorderings. Our goal is to reduce the problem of proving linearizability for all traces of an object to proving linearizability only for traces in a quotient. Therefore, given two traces τ and τ' that are equivalent up to commutativity ($\tau \equiv \tau'$), where for instance, τ would be part of a quotient, an important question is whether the linearizability of τ implies the linearizability of τ'. We show that this holds provided that the reordering allowed by the equivalence $\equiv$ is consistent with a commutativity relation between operations in the specification.

Given a specification S, two operations o_1 and o_2 are S***-commutative*** when $\eta_1 \cdot o_1 \cdot o_2 \cdot \eta_2 \in S$ iff $\eta_1 \cdot o_2 \cdot o_1 \cdot \eta_2 \in S$, for every η_1, η_2 sequences of operations. Given a set of pairs of labels $\text{Swaps} \subseteq \mathcal{L} \times \mathcal{L}$, a linearization point mapping $lp(\tau)$ of a trace τ is ***robust against*** Swaps***-reorderings*** if for every two threads t_1 and t_2, if the linearization points of t_1 and t_2 form a pair in Swaps, then the operations of t_1 and t_2 are S-commutative.

Theorem 3.5. *Let $\tau \equiv \tau'$ be two equivalent traces, such that τ' is obtained from τ by swapping pairs of labels in some set* Swaps. *If τ is linearizable w.r.t. some specification S via a linearization point mapping $lp(\tau)$ that is robust against* Swaps*-reorderings, then τ' is linearizable w.r.t. S.*

The above holds by defining $lp(\tau')$ by $lp(\tau')(t)$ = the index in τ' of the label $lp(\tau)(t)$, for every t.

Theorem 3.5 implies that proving linearizability for an object O reduces to proving linearizability only for the traces in a quotient $\langle\!\lfloor O \rfloor\!\rangle$ of O, provided that the used linearization point mappings are robust against Swaps-reorderings for a set Swaps which is $\langle\!\lfloor O \rfloor\!\rangle$-sufficient (thread renaming does not affect this reduction because specifications are agnostic to thread ids).

4 FINITE ABSTRACT REPRESENTATIONS OF QUOTIENTS

We define finite representations of sets of traces, quotients in particular, which resemble regular expressions and which denote context-free languages over a finite alphabet. The finite alphabet is obtained by projecting out thread ids from labels in a trace. As we show in the evaluation section, scenarios in previous informal proofs of many concurrent objects correspond to components of these expressions, and linearization points can be identified directly within such expressions.

Let Abs be the set of expressions expr defined by the following grammar

$$\mathsf{expr} = \omega \mid \omega_1^n \cdot \mathsf{expr} \cdot \omega_2^n \mid \mathsf{expr}^* \mid \mathsf{expr} + \mathsf{expr} \mid \mathsf{expr} \cdot \mathsf{expr}$$

such that $\omega, \omega_1, \omega_2 \in (\mathsf{A} \cup \mathsf{B} \cup \langle\!\langle b \cdot a \rangle\!\rangle)^*$ are finite sequences of labels, and for every application of the production rule $\omega_1^n \cdot \mathsf{expr} \cdot \omega_2^n$, n is a fresh variable not occurring in expr (this ensures context-free abstractions). Therefore, for every expression in Abs, a variable n is used exactly twice.

Such expressions have a natural interpretation as context-free languages by interpreting *, $+$, and $\cdot$ as the Kleene star, union, and concatenation in regular expressions, and interpreting every $\omega_1^n \cdot \mathsf{expr} \cdot \omega_2^n$ as sequences $\omega_1, \ldots, \omega_1 \cdot [\![\mathsf{expr}]\!] \cdot \omega_2, \ldots, \omega_2$ where the number of ω_1 repetitions on the left of expr's interpretation, denoted as $[\![\mathsf{expr}]\!]$, equals the number of ω_2 repetitions on the right.

We define an interpretation $[\![\mathsf{expr}]\!]$ of expressions expr as sets of *traces*, which differs from the above only in the interpretation of ω, ω^*, and $\omega_1^n \cdot \mathsf{expr} \cdot \omega_2^n$, for finite sequences of labels $\omega, \omega_1, \omega_2$.

Definition 4.1 (Interpretation of an expression). For an expression expr,

$[\![\omega]\!]$	=	$\{t : \omega \mid t \in \mathcal{T}\}$, where $t : \omega$ means that all the labels in ω are associated with the same thread id t,
$[\![\omega^*]\!]$	=	$\{t_0 : \omega, \ldots, t_k : \omega \mid k \in \mathbb{N}, t_0 < \ldots < t_k\}$, sequences of labels associated with increasing thread ids,
$[\![\omega_1^n \cdot \mathsf{expr} \cdot \omega_2^n]\!]$	=	$\{t_0 : \omega_1, \ldots, t_k : \omega_1, [\![\mathsf{expr}]\!], t_k : \omega_2, \ldots, t_0 : \omega_2 \mid k \in \mathbb{N}, t_0 < \ldots < t_k\}$, sequences of labels where the same sequence of increasing thread ids is associated to ω_1 and ω_2 repetitions (in reverse order), respectively,
$[\![\mathsf{expr}^*]\!]$	=	$[\![\mathsf{expr}]\!], \ldots, [\![\mathsf{expr}]\!]$, sequences of repetitions of $[\![\mathsf{expr}]\!]$,
$[\![\mathsf{expr}_1 + \mathsf{expr}_2]\!]$	=	$[\![\mathsf{expr}_1]\!] \cup [\![\mathsf{expr}_2]\!]$, union of interpretations, and
$[\![\mathsf{expr}_1 \cdot \mathsf{expr}_2]\!]$	=	$[\![\mathsf{expr}_1]\!], [\![\mathsf{expr}_2]\!]$, concatenation of interpretations.

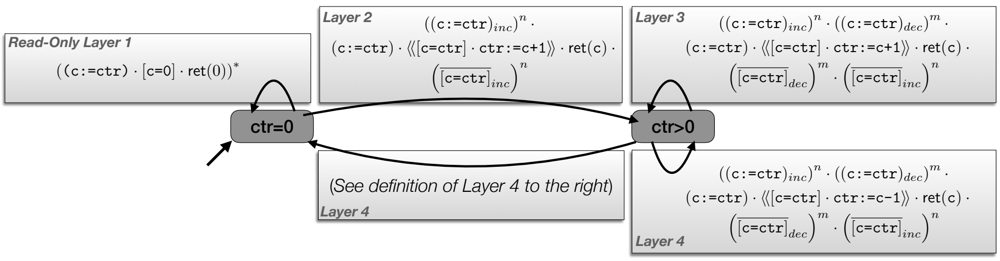

Read-Only Layer 1
$((\texttt{c:=ctr}) \cdot [\texttt{c=0}] \cdot \mathsf{ret}(0))^*$
Layer 2
$((\texttt{c:=ctr})_{inc})^n \cdot$
$(\texttt{c:=ctr}) \cdot \langle\!\langle[\texttt{c=ctr}] \cdot \texttt{ctr:=c+1}\rangle\!\rangle \cdot \mathsf{ret}(\texttt{c}) \cdot$
$\left(\overline{[\texttt{c=ctr}]}_{inc}\right)^n$
Layer 3
$((\texttt{c:=ctr})_{inc})^n \cdot ((\texttt{c:=ctr})_{dec})^m \cdot$
$(\texttt{c:=ctr}) \cdot \langle\!\langle[\texttt{c=ctr}] \cdot \texttt{ctr:=c+1}\rangle\!\rangle \cdot \mathsf{ret}(\texttt{c}) \cdot$
$\left(\overline{[\texttt{c=ctr}]}_{dec}\right)^m \cdot \left(\overline{[\texttt{c=ctr}]}_{inc}\right)^n$
ctr=0
ctr>0
(See definition of Layer 4 to the right)
Layer 4
$((\texttt{c:=ctr})_{inc})^n \cdot ((\texttt{c:=ctr})_{dec})^m \cdot$
$(\texttt{c:=ctr}) \cdot \langle\!\langle[\texttt{c=ctr}] \cdot \texttt{ctr:=c-1}\rangle\!\rangle \cdot \mathsf{ret}(\texttt{c}) \cdot$
$\left(\overline{[\texttt{c=ctr}]}_{dec}\right)^m \cdot \left(\overline{[\texttt{c=ctr}]}_{inc}\right)^n$
Layer 4

Fig. 4. An expression representing a quotient of the Counter. For readability we present it as four sub-expressions called "layers" whose composition with regular expression operators (concatenation, union, star) is represented using an automaton (all states are accepting). The full formal definitions of an example layer—from the quotient expression grammar—is given in Example 5.3. In this figure, for conciseness, we subscript the primitives to indicate whether they were from increment-vs-decrement. Layer 1 represents decrements acting alone and finding the counter to be 0, Layer 2 corresponds to the first successful increment, Layer 3 and Layer 4 represent successful increments and decrements. For Layers 2 – 4, some number x of threads begin to read then a single different thread performs its complete write path, and then all x threads fail their CAS instructions. Technically, Layer 2 is a specialization of Layer 3, by letting $m = 0$. However, treating them as separate layers provides a more refined representation.

For example, in the first case of Def. 4.1, $\{(t : \texttt{x:=v}), (t : \texttt{x++})\} \in [\![\texttt{x:=v} \cdot \texttt{x++}]\!]$. For an expression $(\texttt{x:=r}^n \cdot \texttt{y:=s}^m \cdot \texttt{skip} \cdot \texttt{s:=y+1}^m \cdot \texttt{r:=x+1}^n)$, its interpretation includes traces such as

$$(t_1 : \texttt{x:=r}), (t_2 : \texttt{x:=r}), (t_3 : \texttt{y:=s}), (t_4 : \texttt{skip}), (t_3 : \texttt{s:=y+1}), (t_2 : \texttt{r:=x+1}), (t_1 : \texttt{r:=x+1})$$

Definition 4.2 (Abstractions of quotients). An expression expr $\in$ Abs is called an ***abstraction*** of an object quotient $\langle\!\lfloor O \rfloor\!\rangle$ if $\langle\!\lfloor O \rfloor\!\rangle \subseteq [\![\text{expr}]\!]$.

Example 4.3 (Abstraction of a Quotient of the Counter). An expression representing a quotient of the counter is given in Figure 4. The following trace is in the interpretation of this expression (for readability, we split the trace across lines, with segments labeled by layer names):

$$\begin{array}{ll}
\text{Layer } 2: & t_2 : (c := ctr) \cdot t_3 : (c := ctr) \cdot (t_1 : (c := ctr) \cdot t_1 : \langle\!\langle [c = ctr] \cdot ctr := c+1 \rangle\!\rangle \cdot t_1 : \text{ret}(0)) \cdot \\
& \quad t_3 : \overline{[c = ctr]} \cdot t_2 : \overline{[c = ctr]} \cdot \\
\text{Layer } 3: & t_3 : (c := ctr) \cdot t_2 : (c := ctr) \cdot t_2 : \langle\!\langle [c = ctr] \cdot ctr := c+1 \rangle\!\rangle \cdot t_2 : \text{ret}(1) \cdot t_3 : \overline{[c = ctr]} \cdot \\
\text{Layer } 3: & t_3 : (c := ctr) \cdot t_3 : \langle\!\langle [c = ctr] \cdot ctr := c+1 \rangle\!\rangle \cdot t_3 : \text{ret}(2)
\end{array}$$

Linearizability. Each layer corresponds to linearizing a single *effectful* invocation, *i.e.,* an increment invocation or a decrement invocation when the counter is non-zero, or an arbitrary number of *read-only* invocations, *i.e.,* decrement invocations when the counter is zero.

5 LAYERS: AN INDUCTIVE QUOTIENT LANGUAGE

We show that, for a broad class of objects, we can provide a subclass of quotient abstraction expressions—that we will call *layer expressions*—which, via an inductive argument, reduce reasoning about all executions (and about linearizability) to two-threads. This applies to numerous canonical examples such as Treiber Stack, the Michael-Scott Queue, a linked-list Set, and even the SLS Reservation Queue. For illustrative purposes, we will continue to use the concurrent Counter, whose quotient can also be expressed with layers.

Many lock-free[4] objects rely on a form of optimistic concurrency control where an operation repeatedly reads the shared-memory state in order to prepare an update that reflects the specification and tries to apply a possible update using an atomic read-write. The condition of the atomic read-write checks for possible interference from other threads since reading the shared-memory state. The executions of such objects can be seen as sequences of what we call "layers," each one being a triple consisting of (i) many threads all performing commutative local (*e.g.*, read) actions, (ii) a single non-commutative atomic read-write ARW on the shared state, and (iii) those same initial threads reacting to the ARW with more local commutative actions. For example, incrementing the counter involves a successful cas operation on the shared variable, which leads to other threads' old reads to go down a failure/restart path. In fact, with this layer language one can consider an arbitrary number of control-flow paths executed by an arbitrary number of threads where at most one can contain an atomic read-write. In the remainder of this section we discuss this in detail and then discuss automated discover of layers in Sec. 7.

5.1 Local-vs-Write Paths

For an implementation call $m(\vec{x}) \cdot k_m \in \mathcal{K}$ of a method $m(\vec{x})/\vec{v}$, a *full (control-flow) path* of k_m is a KAT expression k such that $k \leq k_m$ and k contains only primitive actions, tests or ARWs, composed together with $\cdot$ (k contains no $+$ or * constructor). In a representation with control-flow graphs of m's code, k corresponds to a path from the entry point to the exit point. A *path* is any contiguous subsequence k' of a full path k, i.e., there exists (possibly empty) k_1 and k_2 such that $k = k_1 \cdot k' \cdot k_2$. The set of paths of method m is denoted by $\Pi(m)$, and as a straightforward extension, the set of paths of an object O defined by a set of methods m_i with $1 \leq i \leq n$ is defined as $\Pi(O) = \bigcup_{1 \leq i \leq n} \Pi(m_i)$. $\Pi_f(O)$ denotes the subset of *full* paths in $\Pi(O)$.

A primitive action is called *local* when it cannot affect actions or tests executed by another thread (atomic read-writes included), e.g., it represents a read of a shared variable or it reads/writes a memory region that has been allocated but not yet connected to a shared data structure (this region is still *owned* by the thread). Formally, let $[\![a]\!] : (\Sigma_{lo} \times \Sigma_{gl}) \to (\Sigma_{lo} \times \Sigma_{gl})$ and $[\![b]\!] : (\Sigma_{lo} \times \Sigma_{gl}) \to \{\mathit{true}, \mathit{false}\}$ denote the functions defining the semantics of actions $a \in \mathsf{A}$ and tests $b \in \mathsf{B}$. Then, an action $a \in \mathsf{A}$ is *local* iff for every $(\sigma'_l, \sigma'_g) = [\![a]\!](\sigma_l, \sigma_g)$ and every $s \in \mathsf{A} \cup \mathsf{B}$ that occurs in some method implementation, $[\![s]\!](\sigma''_l, \sigma_g) = [\![s]\!](\sigma''_l, \sigma'_g)$, for every local state σ''_l.

A path is called *local* if it contains only local actions, and a *write path*, otherwise. Given a KAT expression k' that represents a path, we use *first*(k') and *last*(k') to denote the first and the last action or test in k', respectively.

Example 5.1. Returning to the counter object O_{ctr}, the full paths are as follows:

$$\texttt{(c:=ctr)} \cdot \overline{\texttt{[c=ctr]}}$$
$$\texttt{(c:=ctr)} \cdot \langle\!\langle \texttt{[c=ctr]} \cdot \texttt{ctr:=c+1} \rangle\!\rangle \cdot \mathrm{ret}(\mathrm{c})$$
$$\texttt{(c:=ctr)} \cdot \texttt{[c=0]} \cdot \mathrm{ret}(0)$$
$$\texttt{(c:=ctr)} \cdot \overline{\texttt{[c=ctr]}}$$
$$\texttt{(c:=ctr)} \cdot \langle\!\langle \texttt{[c=ctr]} \cdot \texttt{ctr:=c-1} \rangle\!\rangle \cdot \mathrm{ret}(\mathrm{c})$$

The first two paths are from k_{inc} and the last three are from k_{dec}. Paths without ARWs consist of only *local* actions, that may read global `ctr`, but they do not mutate any global variables.

5.2 The Language of Layers

We now define layer expressions and discuss how they represent an object's quotient.

Definition 5.2 (Basic Layer Expressions). A *basic layer expression* λ has one of two forms:

[4]Lock-freedom requires that at least one thread makes progress, if threads are run sufficiently long. A slow/halted thread may not block others, unlike when using locks.

- *local layer*: $(k_l)^*$ where k_l is a *local* path in $\Pi(O)$.
- *write layer*: $\left((\overleftarrow{k}_1)^{n_1} \cdot (\overleftarrow{k}_2)^{n_2} \cdots (\overleftarrow{k}_N)^{n_N}\right) \cdot k_w \cdot \left((\overrightarrow{k}_N)^{n_N} \cdot (\overrightarrow{k}_{N-1})^{n_{N-1}} \cdots (\overrightarrow{k}_1)^{n_1}\right)$, where
  (1) k_w is a write path in $\Pi(O)$,
  (2) for each $j \in [1, N]$, $\overleftarrow{k}_j \cdot \overrightarrow{k}_j$ is a *local* path in $\Pi(O)$ and the prefix and suffix are each repeated n_j times,
  (3) $last(\overleftarrow{k}_j)$ and $first(\overrightarrow{k}_j)$ do not commute with respect to the ARW in k_w.

The first type, *local layers*, represent unboundedly many threads executing a local path k_l. Since each instance of the path is local, they all commute with each other, so the interpretation puts them into a single, canonical order which follows the increasing order between their thread ids (by the interpretation of $*$ in quotient expressions; see Def. 4.1).

The second type, *write layers*, represents an interleaving where threads execute n_j read-only prefix $\overleftarrow{k}_j$ of paths (in a canonical, serial order), then a different thread executes a non-local path k_w, and then n_j corresponding suffixes $\overrightarrow{k}_j$ occur, finishing their iteration reacting to the write of k_w. Again, the interpretation $[\![\lambda]\!]$ of a write layer associates these KAT action labels with increasing thread ids. Prefixes and suffixes of local paths can be assumed to execute serially as in the first type of layer. The non-commutativity constraint ensures that such an interleaving is "meaningful", *i.e.,* it is not equivalent to one in which complete paths are executed serially.

A ***layer expression*** is a collection of basic layer expressions, combined in a regular way via $\cdot$, $+$, or $*$ (defined in Sec. 4). That is, a layer expression represents complete traces as sequences of layers.

Example 5.3. The expression given in Fig. 4 representing a quotient of the Counter is a layer expression. It combines a single read-only layer with other three write layers. One layer λ_{inc} pertains to the increment write path, along with the local paths that fail their CAS attempts. Here, we consider full paths. This basic expression λ_{inc} is:

$$\lambda_{inc} \equiv \left[\left(\odot \begin{array}{l} (\texttt{c:=ctr})_{inc} \\ (\texttt{c:=ctr})_{dec} \end{array} \right) \cdot (\texttt{c:=ctr}) \cdot \langle\!\langle [\texttt{c=ctr}] \cdot \texttt{ctr:=c+1} \rangle\!\rangle \cdot \texttt{ret(c)} \cdot \left(\odot \begin{array}{l} \overline{[\texttt{c=ctr}]}_{dec} \\ \overline{[\texttt{c=ctr}]}_{inc} \end{array} \right) \right]^{\circledast}$$

This layer interleaves the write path between prefixes/suffixes of the two local paths. We subscript the primitives to indicate whether they were from increment-vs-decrement. The *first* and *last* actions/tests do not commute with the interleaved writer's ARW.

Support of a Layer. The *support* of a basic layer expression λ, denoted by $supp(\lambda)$, is defined as a set of KAT expressions where a single prefix/suffix local path is concretized to a single occurrence, and interleaved with the write path. Intuitively, the support of a write layer characterizes all of the pair-wise interference by representing interleavings of two paths executed by different threads.

Definition 5.4. For basic layer expression λ, $supp(\lambda)$ is defined as:

- If λ is a local layer $\lambda = (k_l)^*$, then $supp(\lambda) = \{k_l\}$.
- If λ is a write layer $\lambda = \left((\overleftarrow{k}_1)^{n_1} \cdot (\overleftarrow{k}_2)^{n_2} \cdots (\overleftarrow{k}_N)^{n_N}\right) \cdot k_w \cdot \left((\overrightarrow{k}_N)^{n_N} \cdot (\overrightarrow{k}_{N-1})^{n_{N-1}} \cdots (\overrightarrow{k}_1)^{n_1}\right)$, then $supp(\lambda) = \{\overleftarrow{k}_j \cdot k_w \cdot \overrightarrow{k}_j \mid j \in [1, n]\}$.

Example 5.5. For Layer 3 in Fig. 4 involving the increment write path $k_w = (\texttt{c:=ctr}) \cdot \langle\!\langle [\texttt{c=ctr}] \cdot \texttt{ctr:=c+1} \rangle\!\rangle \cdot \texttt{ret(c)}$, $supp(\text{Layer 3}) = \{(\texttt{c:=ctr})_{inc} \cdot k_w \cdot \overline{[\texttt{c=ctr}]}_{inc}, (\texttt{c:=ctr})_{dec} \cdot k_w \cdot \overline{[\texttt{c=ctr}]}_{dec}\}$. Here there are only two elements of the support, the first being a local path through increment and the second being a local path through decrement.

The paths $\Pi(\lambda)$ of a basic layer expression λ are defined from its support: (1) if λ is a local layer, then $\Pi(\lambda) = supp(\lambda)$, and (2) if λ is a write layer, then $\{k_w, \overleftarrow{k}_j \cdot \overrightarrow{k}_j\} \subseteq \Pi(\lambda)$ iff $\overleftarrow{k}_j \cdot k_w \cdot \overrightarrow{k}_j$ is included in $supp(\lambda)$. The paths $\Pi(\text{expr})$ of a layer expression expr is obtained as the union of $\Pi(\lambda)$ for every basic layer expression λ in expr.

5.3 Proof Methodology with Two-Thread Reasoning

Recall that layer expressions represent languages of traces so we now ask whether a given expression is an abstraction of an object's quotient (Def. 4.2). That is: whether each execution ρ of an object is equivalent to some execution $\rho' \equiv \rho$, where the trace of ρ' is in the interpretation of the expression.

Interestingly, this can be done by considering only two threads at a time, since local paths do not affect the feasibility of a trace. Therefore, it is sufficient to focus on interleavings between a *single* local or write path k (on a first thread) and a sequence $\vec{k}_w$ of (possibly different) write paths (on a second thread), and show that they can be reordered as a sequence of layers, i.e., k executes in isolation if it is a write path, and interleaved with at most one other write path in $\vec{k}_w$, otherwise (it is a local path). Applying such a reordering for each path k while ignoring other local paths makes it possible to group paths into layers. The reordering must preserve a stronger notion of equivalence defined as follows: two executions ρ and ρ' are *strongly equivalent* if they are $\equiv$-equivalent, they start and resp., end in the same configuration, and they go through the same sequence of shared states modulo stuttering. This notion of equivalence guarantees that any local path enabled in the context of an arbitrary interleaving between k and $\vec{k}_w$ remains enabled in the context of an interleaving where for instance, k executes in isolation. A more detailed proof for the following theorem is given in the extended version [Enea et al. 2023].

THEOREM 5.6. *Let O be an object defined by a set of methods m_i with implementations call $m_i(\vec{x}) \cdot k_{m_i} \in \mathcal{K}$. A layer expression* $\text{expr} = (\lambda_1 + \ldots + \lambda_n)^*$ *is an abstraction of a quotient of O if*

- *the layers cover all statements in the implementation:* $\Pi(\text{expr}) \subseteq \Pi(O)$ *and for each primitive action, test or ARW k_p in k_{m_i} for some i, there exists a path in* $\Pi(\text{expr})$ *which contains k_p,*
- *for every path $k \in \Pi(\text{expr})$ and every execution ρ of O starting in a reachable configuration that represents[5] an interleaving $k \;||\; \vec{k}_w$, where $\vec{k}_w$ is a sequence of write paths in* $\Pi(\text{expr})$*,*
  - *Write Path Condition (WPC): if k is a write path, there is an exec. ρ' of O s.t. ρ' is strongly equivalent to ρ, and ρ' represents a write path sequence $\vec{k}^1_w \cdot k \cdot \vec{k}^2_w$ where $\vec{k}_w = \vec{k}^1_w \cdot \vec{k}^2_w$,*
  - *Local Path Condition (LPC): if k is a local path, there exists an execution ρ' of O such that ρ' is strongly equivalent to ρ and*
    - *ρ' represents a path sequence $\vec{k}^1_w \cdot k \cdot \vec{k}^2_w$ where $\vec{k}_w = \vec{k}^1_w \cdot \vec{k}^2_w$ (k executes in isolation) and k is the support of a local layer λ_j, $1 \leq j \leq n$, or*
    - *a sequence $\vec{k}^1_w \cdot k^1_l \cdot k_w \cdot k^2_l \cdot \vec{k}^2_w$ where $\vec{k}_w = \vec{k}^1_w \cdot k_w \cdot \vec{k}^2_w$ and k_w is a write path (k interleaves with a single write path k_w), and $k^1_l \cdot k_w \cdot k^2_l \in supp(\lambda_j)$ for some write layer λ_j, $1 \leq j \leq n$.*

Example 5.7 (Counter layers via two-thread reasoning). We now proceed to show that the *starred union* of the basic layer expressions defined in Fig. 4 is an abstraction of a quotient. Concerning WPC, a write path is of the form `(c:=ctr)` $\cdot$ ⟪`[c=ctr]` $\cdot$ `ctr:=c+1`⟫ $\cdot$ `ret(c)`. Such paths can be reordered to execute in isolation because the ARW is enabled only if the counter did not change its value since the read, and therefore, the read `c:=ctr` can be reordered after any step of another

[5]An execution ρ represents an interleaving $k \;||\; \vec{k}_w$ if it interleaves two sequences of steps labeled with symbols in k and $\vec{k}_w$, respectively (in the same order). An execution ρ represents a path sequence $\vec{k}$ when it is a sequence of steps labeled with symbols in $\vec{k}$ (in the same order).

thread that may occur until the ARW. Also, the return action is local and can be reordered to occur immediately after the ARW. LPC holds because any "late" CAS failure (that occurs after more than one successful CAS) would also fail if moved to the left (as explained in Example 3.4).

5.4 Automaton Representation of Layer Quotients

A layer expression comprised simply of a starred union of basic layer expressions is not always appealing since some layers are not enabled from some configurations. For instance, as shown in Figure 4 for the Counter, the read-only "decrement returning 0" layer cannot occur after one successful increment layer. (In formal notation, layer λ_{dec0} of O_{ctr} in Example 5.3 is enabled only when `ctr` is 0.) In other words, the starred starred union composition of layers can be refined further to enforce certain orders in which layers can occur, by taking into account reachability.

We therefore describe a more convenient representation as a *layer automaton*, in which the automaton states represent abstractions (sets) of concrete configurations in executions (as defined in Sec. 2) and the transitions are labeled by basic layer expressions. Another example of such an automaton was seen for the Michael-Scott queue in Fig. 1 in Sec. 1. Briefly, the control states correspond to the configurations of the objects (*e.g.,* , whether the MSQ is empty, tail is lagged, etc.), and the transitions are labeled by basic layer expressions (*e.g.,* , the "Dequeue Succeed" layer from Fig. 1, in which one thread succeeds a CAS on the `head` pointer and other threads fail their CAS). These layer automata are a convenient representation of the quotient and, as shown in Sec. 7, we can derive candidate layer quotients represented as layer automata automatically from source code.

Definition 5.8 (Layer automaton). Given an object O, a *layer automaton* is a tuple $\mathcal{A} = (Q, Q_0, \Lambda, \delta)$ where Q is a finite set of states representing abstractions (sets) of configurations of O, $Q_0 \subseteq Q$ is the set of initial states, and $\delta \subseteq Q \times 2^{\Lambda} \times Q$ is a set of transitions labeled with basic layer expressions (elements of Λ) with the constraint that an edge $q \xrightarrow{\alpha} q'$ can only be one of two types:

(1) Unique self-loop: $\alpha = \lambda_1 \cdots \lambda_n$ is a sequence of $n \geq 1$ local layers, $q' = q$, and there are no other self-loops $q \xrightarrow{\alpha'} q$.
(2) Single write layer edges: $\alpha = \lambda$ is a single write layer.

The ***interpretation*** of the automaton, denoted by $[\![\mathcal{A}]\!]$, as a layer expression is defined as expected, except that the label of a self-loop is not starred. For instance, the interpretation of an automaton consisting of a single state q and self-loop $q \xrightarrow{\alpha} q$ is defined as α instead of α^*.

Theorem 5.9. *Given an object O and a layer automaton $\mathcal{A} = (Q, Q_0, \Lambda, \delta)$, the layer expression $[\![\mathcal{A}]\!]$ is an abstraction of a quotient of O if*

- *the starred union of the basic layer expressions labeling transitions of $\mathcal{A}$ is an abstraction of a quotient of O (Theorem 5.6),*
- *every initial configuration of O is represented by some abstract state in Q_0, and every reachable configuration is represented by some abstract state in Q,*
- *for every layer λ in $[\![\mathcal{A}]\!]$, if there exists an execution ρ representing λ from a reachable configuration C to a configuration C', then $\mathcal{A}$ contains a transition $q \xrightarrow{\alpha'} q$ where q is an abstraction of C and q' is an abstraction of C'.*

The automaton in Fig. 1 is a layer automaton for the MSQ (see Section 6.1 for more details).

Corollary 5.10. *(To Thm. 3.5) If a layer expression* expr *is an abstraction of a quotient and there is a linearization point mapping for every trace in* $[\![\text{expr}]\!]$ *that is robust against re-ordering, then the object is linearizable.*

6 EVALUATION: VERIFYING CONCURRENT OBJECTS

As discussed in Sec. 1, our goal is to provide a formal foundation for the scenario-based linearizability correctness arguments found in the distributed computing literature. To evaluate whether quotients serve that purpose, we examined several diverse and challenging concurrent objects, listed below.

Concurrent Object	**Quotient**	**Features**
Atomic counter	Sec. 2	simple cas loop
Michael and Scott [1996] queue	**Sec. 6.1**	many cas, cleanup helping
Scherer III et al. [2006] queue	**Sec. 6.2**	synchronous, mult. writes, LP helping
[Treiber 1986]'s stack	**Sec. 6.3**	simple cas loop
Hendler et al. [2004] stack	**Sec. 6.3**	elimination, submodule, LP helping
Harris et al. [2002] RDCSS	**Sec. 6.4**	mult. cas steps, phases
Herlihy and Wing [1990] queue	**Sec. 6.5**	future-dependent LPs
O'Hearn et al. [2010] set	Ext. Ver.	lock-free traversal

For each object, we (i) determine whether quotients can be used for verification and (ii) revisit the scenario-based correctness arguments given by the object's authors and compare those arguments to the quotient. We discuss the quotients of most in this section (with bold **Sec 6._** in the **Quotient** column); further detail can be found in the appendix of the extended version [Enea et al. 2023].

Results summary. As we show, all above algorithms can be captured with quotient expressions. These expressions (i) capture the diverse features/complexities of these algorithms (per the **Features** column), (ii) provide a succinct, formal foundation for the scenario-based arguments used by those objects' authors, (iii) organize unbounded interleavings into a form more amenable to reasoning, (iv) make explicit the relationship between implementation-level contention/interference and ADT-level transitions, and (v) provide a scenario proof for HWQ which did not have scenario arguments.

6.1 The Michael/Scott Queue

Recall the implementation of MSQ, stored as a linked list from global pointers Q.head and Q.tail, and manipulated as follows. (Some local variable definitions omitted for lack of space.)

```
int enq(int v){ loop {
 node_t *node=...;
 node->val=v;
 tail=Q.tail;
 next=tail->next;
 if (Q.tail==tail) {
  if (next==null) {
   if (CAS(&tail->next,
            next,node))
      ret 1;
} } } }
```

```
int deq(){ loop {
 int pval;
 head=Q.head;tail=Q.tail;
 next=head->next;
 if (Q.head==head) {
  if (head==tail) {
   if (next==null) ret 0;
  } else {
    pval=next->val;
    if (CAS(&Q->head,
            head,next))
      ret pval;
  } } } } }
```

Factored out tail advancement:
(see notes below)

```
adv(){ loop {
 tail=Q.tail;
 next=tail->next;
 if (next!=null){
   if (CAS(&Q->tail,
      tail,next))
     ret 0;
 }
} }
```

Values are stored in the nodes between Q.head and Q.tail, with enq adding new elements to the Q.tail, and deq removing elements from Q.head. During a successful CAS in enq, the Q.tail->next pointer is changed from null to the new node. However, this new item cannot be dequeued until adv advances Q.tail forward to point to the new node. A deq on an empty list (when Q.head=Q.tail) returns immediately. Otherwise, deq attempts to advance Q.head and, if success, returns the value in the now-omitted node. The original MSQ implementation includes the adv CAS inside enq and deq iterations. We have done this for expository purposes and it is not necessary. As we will see in Sec. 6.2, the SLS queue performs this tail (and head) advancing directly in the enqueue/dequeue method implementation.

Quotient. The layer automaton that abstracts a quotient of MSQ, mentioned briefly in Sec. 1, is shown in Fig. 1. The automaton states track whether `Q.tail=Q.head` and whether `Q.tail->next` is `null`, in rounded dark boxes. Edges are labeled with layers (discussed below), defined to the right in Fig. 1. The write operations in those layers induce the automaton state changes as shown by the various edges between automaton states. For example, the Dequeue Succeed layer can move from automaton state q_2 to q_1. The three layers of the MSQ characterize three forms of interference:

The Dequeue Succeed layer occurs when a dequeue thread successfully advances the `Q.head` pointer, causing concurrent dequeue CAS attempts to fail, as well as dequeue threads checking on Line 5 whether `Q.head` has changed. (We abbreviate local paths using line numbers rather than KAT expressions.)

The Advancer Succeed layer occurs when an advancer moves forward the `Q.tail` pointer, causing concurrent advancer CAS attempts to fail, and causing concurrent `enq` threads to find `Q.tail` changed on Line 6.

The Enqueue Succeed layer occurs when an `enq` thread successfully advances the `Q.tail` pointer, causing concurrent `enq` threads to fail.

Naturally, some edges are not enabled. For example, there is no edge from q_1 to q_2, because the latter is not reachable from the former via a single write path/layer. Also, while there are outbound edges from q_1, there is no layer involving a `deq` write operation (since the queue is empty). Some non-local layers self-loop, such as the Dequeue Succeed layer self-loop at q_4. There are also four *local* layers that self-loop. These involve local paths that return (*e.g.,* Read Only Layer 1 where `deq` returns because the queue is empty) or paths that loop while waiting (*e.g.,* Read Only Layer 3 where `enq` awaits the advancer thread).

Theorem 6.1. *The above layer automaton is an abstraction of a quotient for Michael-Scott Queue.*

Proof: Proof by the methodology of Def. 5.6.

The WPC condition requires that all write paths (that include successful CASs) can be reordered to execute in isolation. This is a direct consequence of the semantics of a successful CAS which checks that the value did not change since the last read of the written location. The `deq` successful CAS on `Q.head` insures that `Q.head` did not change since it was read at Line 3, which also means that its `next` pointer did not change (this pointer is written only once in `enq()` for every node in the list). Therefore all actions on the `deq` path that includes the successful CAS can be reordered to execute together at the place of reading `Q.tail`. Similarly the `enq` successful CAS ensures that the actions between Line 5 and Line 8 can be reordered to occur together. Then, since the value of `Q.tail` could not have changed without `Q.tail->next` first having been changed, Lines 2-4 can also be reordered to occur together with the rest of the actions on this path. The case of the `adv` write path is similar.

The LPC condition follows from the fact that CAS operations always change the value so it is always possible to move a late "failing" CAS to the left so that it occurs after the first successful CAS following the previous reads in the same iteration. ■

Theorem 6.2. *The Michael-Scott Queue is linearizable.*

Proof: We show that the traces in the quotient are linearizable via a linearization-point mapping which is robust against reorderings. Given a trace in the quotient (represented by the automaton in Fig. 1), the linearization points are the successful CAS operations in the {Dequeue,Advancer} Succeed layers (also in **bold** in the Fig. 1 layer definitions), as well as the action corresponding to Line 7 in `deq()` which occurs in Read-Only Layer 1. The successful CAS operations are linearization points of dequeues returning some enqueued value and enqueues, respectively, and Line 7 is the linearization point of a dequeue returning empty. The validity of these linearization points can be proved by induction on the number of layers. The induction hypothesis will relate the last configuration of the quotient execution with a queue ADT state that is the sequence of elements reachable from `Q.head`. For instance, the successful CAS in the Dequeue Succeed layer will remove the first element in such a sequence which by the induction hypothesis is the oldest element in the queue.

By the proof of the quotient's completeness (Theorem 6.1), successful CAS operations are never reordered. The only linearization point labels that can be reordered are those corresponding to Line 7 in `deq()` for a dequeue returning empty. It is easy to see that dequeues returning empty commute in the queue specification, which implies that the above linearization-point mapping is robust against a set of reorderings which is sufficient for this quotient. ∎

Comparison with the Authors' Proof. We evaluated the quotient by comparing with the correctness arguments from Herlihy and Shavit [2008]. For lack of space, the following table gives example elements of the correctness argument/proof from Herlihy and Shavit [2008], and identifies where they occur in the quotient proof (see [Enea et al. 2023] for more details).

Proof Element	Herlihy and Shavit [2008]	**Quotient Proof**
ADT states	"queue is nonempty," "tail is lagged"	ADT states, e.g. (`Q.tail=Q.head` ∧ `Q.tail->next` ≠ `null`)
Concurrent threads	"some other thread"	Superscripting $(...)^n$
Event order	"only then"	Arcs in the quo automaton
Thread-local step seq.	"reads tail, and finds the node that appears to be last (Lines 12–13)"	Layer paths, e.g., enq:2-6
Linearization pts.	"If this method returns a value, then its linearization point occurs when it completes a successful [CAS] call at Line 38, and otherwise it is linearized at Line 33."	The successful CAS in the Dequeue Succeed Layer or Read-Only Layer 1

The layer quotient and, especially, the layer automaton helps make the Herlihy and Shavit [2008] proof more explicit, without sacrificing the organization of the proof, for a few reasons. First, all of the important ADT states are explicitly identified. Second, it can be determined, from each of them, which layers are enabled as well as the target ADT states that are reached after each such layer transition. This ensures that all cases are considered. Finally, *linearization points* are explicit in the layer quotient, occurring once with each layer transition.

6.2 The SLS Synchronous Reservation Queue

The Scherer III et al. [2006] (SLS) queue builds on MSQ, but has some complications: queue operations are synchronous (blocking), a single invocation can involve multiple sequentially composed write paths that necessitate different layers, and linearization points must account for dequeuers arriving before their corresponding enqueuer.

Implementation. Like MSQ, SLS has paths that read the `head` or `tail` pointer and subsequent pointers, perform read validations and then attempt a CAS. Also like MSQ, enqueuers arriving at an empty list (or list of items), attempt to append *item* nodes (and then try to advance the tail pointer). Dequeuers arriving at a list of items, attempt to swap item node contents for null (and then try to advance the head pointer).

SLS then has some further complexities. Dequeuers arriving at an empty list (or list of reservation nodes) attempt to append *reservation* nodes (and attempt to advance tail). Enqueuers arriving at a list of reservations, attempt to *fulfill* those reservations by swapping null for an item (and attempt

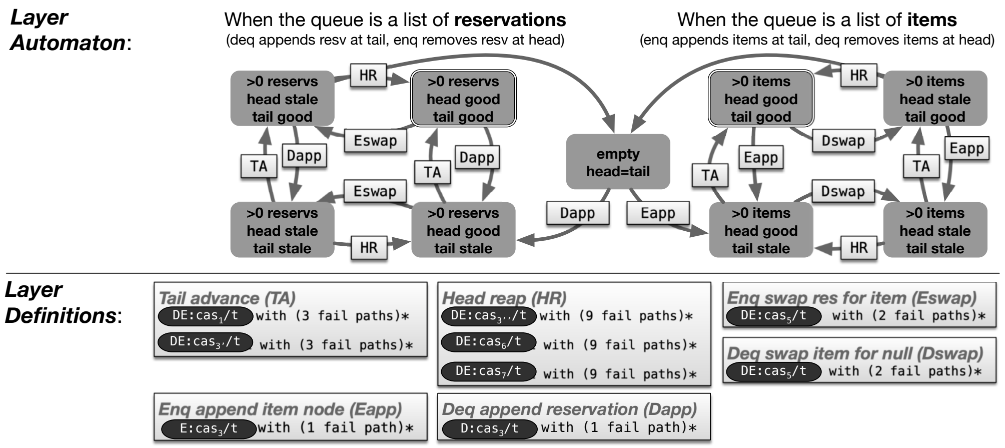

Layer Automaton:
When the queue is a list of reservations
(deq appends resv at tail, enq removes resv at head)
When the queue is a list of items
(enq appends items at tail, deq removes items at head)
>0 reservs head stale tail good
>0 reservs head good tail good
>0 reservs head stale tail stale
>0 reservs head good tail stale
empty head=tail
>0 items head good tail good
>0 items head stale tail good
>0 items head good tail stale
>0 items head stale tail stale
HR
Eswap
TA
Dapp
Eapp
Dswap
Layer Definitions:
Tail advance (TA)
DE:cas1/t with (3 fail paths)*
DE:cas3'/t with (3 fail paths)*
Head reap (HR)
DE:cas3''/t with (9 fail paths)*
DE:cas6/t with (9 fail paths)*
DE:cas7/t with (9 fail paths)*
Enq swap res for item (Eswap)
DE:cas5/t with (2 fail paths)*
Deq swap item for null (Dswap)
DE:cas5/t with (2 fail paths)*
Enq append item node (Eapp)
E:cas3/t with (1 fail path)*
Deq append reservation (Dapp)
D:cas3/t with (1 fail path)*

Fig. 5. Layer automaton for the synchronous SLS queue. Layers' acronyms and their definitions are given in the lower half of the figure. For conciseness, layer definitions do not split the prefix/suffix of the read paths.

to advance head). The list never contains both items and reservations; when the list becomes empty it can then transition from an item list to a reservation list (or vice-versa). Finally, SLS is *synchronous:* dequeuers with reservations *block* until those reservations have been fulfilled and enqueuers with items *block* until those items have been consumed. (For the sake of comprehensiveness, the implementation is in the extended version [Enea et al. 2023], but not necessary for a general understanding.) As noted, unlike MSQ where paths have at most 1 write operation, a single SLS invocation can perform multiple write operations (*e.g.*, a dequeue path inserting a reservation, advancing tail, awaiting fulfillment, advancing head). Despite conceptual simplicity, the implementation is non-trivial with many restart paths when validations or CAS operations fail.

Quotient. The quotient expression for the SLS queue is depicted as a layer automaton in Fig. 5. In the upper portion, the automaton *states* differentiate between whether the queue is empty or whether the queue consists of reservations (left hand region) or of items (right hand region). In each of those regions, it is relevant as to whether the head pointer is stale or not, as well as whether the tail pointer is stale or not. When the queue is a list of reservations, the head or tail could be stale (hence four states) and similar when the queue is a list of items.

The *basic layers* of the quotient expression are defined at the bottom of Fig. 5. The black circles (*e.g.*, DE:CAS$_\ell$/t) represent a write path in which a Dequeuer or Enqueuer has successfully performed a CAS at some program location ℓ. Along with the write path, we simply summarize the number of competing read-only paths, which are star-iterated. Two layers are enq/deq-agnostic: advancing the tail pointer in TA and advancing the head pointer (and "reaping" the head node) in HR. These helping operations happen in many places in the code, with corresponding read-only "_f" failure paths. Enqueue can either append an item node (Eapp) when in the RHS states of the automaton or else swap an item into a reservation node (Eswap) in the LHS. These layers have a single CAS operation (*e.g.*, E:CAS$_5$/t) along with read-only paths where concurrent competing threads fail. The dequeue layers Dapp and Dswap are similar.

Finally, these (context-free) basic layer expressions are connected into an overall expression, represented here as an automaton or (below) as a star-/plus-/or-combination of layer expressions.

Theorem 6.3. *The SLS queue is linearizable.*

Proof: We associate linearization points with layers: Dswap is an LP for dequeue, Eapp is an LP for enqueue, and Eswap is an LP for a combination of an enqueue followed by a dequeue. Next, we project the linearization points out of the quotient to obtain simply $(E \cdot D)^* \cdot (E^* + D^*)$. Combining this with a lemma that this expression is an abstraction of the quotient, we obtain that all executions in the quotient meet the sequential spec. of a queue. This linearization point mapping is also robust because successful CASs (linearization points) do not have to be swapped in order to prove the completeness of the quotient. (Detail in the extended version [Enea et al. 2023].) ■

Comparison with the Authors' Proof. We evaluated the SLS quotient expression by revisiting the authors' proof in Scherer III et al. [2006]. Line numbers in the authors' quotes below refer to a reproduction of the source code given in in the extended version [Enea et al. 2023]. For lack of space, some discussion of the authors' quotes can be found in the extended version [Enea et al. 2023].

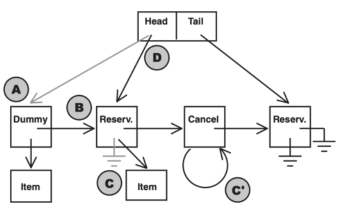
The authors split the enqueue operation into two linearization points: a "reservation linearization point" and a later "follow up linearization point," so that synchronous, blocking enqueue implementations are a single reservation LP and then repeated follow-up LPs (as if the client is repeatedly checking whether the operation has completed).

> *[Regarding enqueue,] the reservation linearization point for this code path occurs at line [...] when we successfully insert our offering into the queue* – Scherer III et al. [2006]

This prose describes a scenario, (i) identifying an alleged linearization point at `E:cas`$_3$`/t`, involving a specific change to shared memory (a CAS on the tail's next pointer), and (ii) identifying the important ADT state transition (inserting an offer node into the queue). This scenario is formalized by the Eapp layer in the quotient expression. The successful CAS `E:cas`$_3$`/t` in Eapp is the linearization point, with competing concurrent threads abstracted away by the starred fail path expression, and the state transition is given in the automaton as the downward Eapp-labeled arcs in the righthand region of the automaton. The scenario and LP for dequeue on a list of reservation nodes is symmetric, and represented in the quotient expression as layer Dapp involving `D:cas`$_3$`/t` and competing fail path.

The quotient expression makes the interaction between LPs and ADT states more explicit (*e.g.*, through *LP*-marked layers) and comprehensive (*e.g.*, the authors do not discuss the 9 different automaton ADT states and which transitions are possible from each). The quotient expression can be seen as an abstract view of an implementation of the sequential specification.

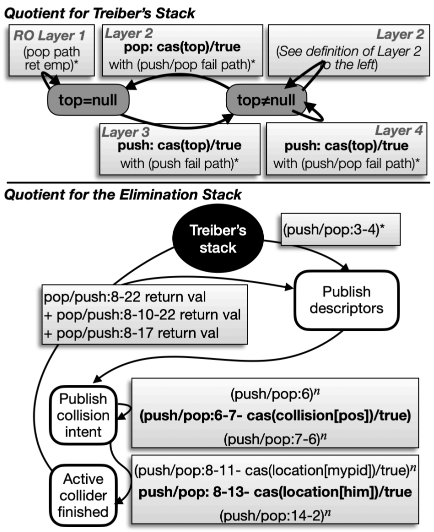
> *The other case occurs when the queue consists of reservations (requests for data), and is depicted [to the right]. In this case, after originally reading the head node (step A), we read its successor (line [...]/step B) and verify consistency (line [...]). Then, we attempt to supply our data to the head-most reservation (line [...]/C). If this succeeds, we dequeue the former dummy node ([...]/D) and return*



This prose again indicates important mutations (*e.g.*, swapping the node's contents pointer), ADT state changes (*e.g.*, supplying data) and that the head dummy node needs to be advanced. These memory mutations and state changes are explicit in the quotient expression. For example, Eswap performs a memory CAS and makes a ADT state transition. The staleness of the head is also captured directly in the ADT states and the HR layers' transitions. The authors' prose also discusses failure paths (see [Enea et al. 2023]) and retry, which are also captured in the layer definitions.

```
1 void push/pop(descriptor p){ while(1) {
2   one iteration of Treiber stack
3   location[mytid] = p;
4   pos = nondet();
5   do { him = collision[pos]
6   } while (!CAS(&collision[pos], him, mytid))
7   if him != NULL {
8      q = location[him]
9      if ( q != NULL & q.id = him & p.op != q.op ) {
10        if (CAS(&location[mytid],p,NULL)) {
11           if ( CAS (&location[him], q, p/NULL) )
12              return NULL/q.input
13           else continue
14        } else {
15           val = NULL/location[mytid].input;
16           location[mytid] = NULL;
17           return val
18  } } }
19  if (!CAS(&location[mytid],p,NULL)) {
20     val = NULL/location[mytid].data;
21     location[mytid] = NULL;
22     return val
23 }}  }
```

(a) Elimination Stack source code


Quotient for Treiber's Stack
RO Layer 1
(pop path ret emp)*
Layer 2
pop: cas(top)/true
with (push/pop fail path)*
Layer 2
(See definition of Layer 2 to the left)
top=null
top≠null
Layer 3
push: cas(top)/true
with (push fail path)*
Layer 4
push: cas(top)/true
with (push/pop fail path)*
Quotient for the Elimination Stack
Treiber's stack
(push/pop:3-4)*
pop/push:8-22 return val
+ pop/push:8-10-22 return val
+ pop/push:8-17 return val
Publish descriptors
Publish collision intent
(push/pop:6)^n
(push/pop:6-7- cas(collision[pos])/true)
(push/pop:7-6)^n
Active collider finished
(push/pop:8-11- cas(location[mypid])/true)^n
push/pop: 8-13- cas(location[him])/true
(push/pop:14-2)^n

(b) Stack Quotients

Fig. 6. Elimination Stack

Summary. The layer quotient expression/automaton provides a succinct formal foundation for the correctness arguments of Scherer III et al. [2006], capturing the authors' discussions of LPs, ADTs, impacts of writes, CAS contention, etc.

6.3 The Hendler et al. Elimination Stack

The Elimination Stack of Hendler et al. [2004] is difficult because the linearization point of some invocation can happen in another (threads can awake to find they were linearized earlier) and it uses a submodule: Treiber's stack [Treiber 1986].

We first show the Treiber's stack quotient, and then build elimination on top. Since Treiber's stack is simple, we explain only the basics here, with more detail in the extended version [Enea et al. 2023]. The implementation of push prepares a new node and then attempts a CAS to swing the top pointer, while pop attempts to advance the top pointer and return the removed node's value. The quotient for Treiber's stack is shown in the upper right of Fig. 6 and is similar to the counter, but with ADT states tracking emptiness (rather than non-zeroness) and CAS contention on the top pointer (rather than the counter cell). There is one read-only layer for a pop and an empty stack, and other layers involve one successful CAS with failed competing CAS attempts. See [Enea et al. 2023] for more detail, as well as a lemma proving that this layer automaton is an abstraction of the quotient.

The Elimination Stack, listed in Fig. 6(a), augments Treiber's stack with a protocol for "colliding" push and pop invocations so that the push passes its input directly to the pop without affecting the underlying data structure. An invocation starts this protocol after performing a loop iteration in Treiber's stack and failing (due to contention on top). The protocol uses two arrays: (1) a `location` array indexed by thread ids where a push or pop invocation publishes a descriptor tuple (`op`,`id`,`input`) with fields `op` for the type of invocation (push or pop), `id` for the id of the invoking thread, and `input` for the input of a push operation, and (2) a `collision` array indexed by arbitrary integers which stores ids of threads announcing their availability to collide.

Each invocation starts by publishing their descriptor in the `location` array (line 3). Then, it reads a random cell of the `collision` array while also trying to publish their id at the same index using a CAS (lines 4–6). If it reads a non-NULL thread id, then it tries to collide with that thread. A successful collision requires 2 successful CASs on the `location` cells of the two threads (we require CASs because other threads may compete to collide with one of these two threads): the initiator of the collision needs to clear its cell (line 10) and modify the cell of the other thread (line 11) to pass its input if the other thread is a pop. The first CAS failing means that a third thread successfully collided with the initiator and the initiator can simply return (lines 15–17). Failing the second CAS leads to a restart (line 13). Succeeding the second CAS means there has been a successful collision and the thread returns, returning null for a push and otherwise using the descriptor to obtain the popped value (line 11). If the invocation reads a NULL thread id from `collision`, then it tries to clear its cell before restarting (line 19). If it fails, then as in the previous case, a collision happened with a third thread and the current thread can simply return (line 20–22).

Quotient. We use the automaton in the lower right of Fig. 6 to describe a sound abstraction of the quotient. Layers of Treiber's stack interleave with layers of the collision protocol (some components are not exactly layers as in Definition 5.2, but quite similar). Executions in the quotient *serialize* collisions and proceed as follows: (1) some number of threads publish their descriptor and choose a cell in the `collision` array, (2) some number of threads publish their id in the `collision` array (there may be more than one such thread – note the self-loop on the "Publish collision intent" state), (3) some number of threads succeed the CAS to clear their `location` cell but only one succeeds to also CAS the `location` cell of some arbitrary but fixed thread `him` and return, and (4) the thread `him` returns after possibly passing the tests at line 7 or 9. (Note that, for succinctness, we have combined push/pop into the same method, which also makes the automaton succinct. The code and corresponding automaton could also have been written in a more verbose way where the bottommost layer is replaced with two layers: (1) a layer where a push's successful CAS takes with it a corresponding pop, and (2) a layer where a pop's successful CAS takes with it a corresponding push. For succinctness, we have combined those layers using the "push/pop" notation.) We emphasize that collisions happen in a serial order, i.e., at any point there is exactly one thread that succeeds on both CASs required for a collision and immediately after the collided thread returns (publishing descriptors or collision intent interleaves arbitrarily with collisions).

THEOREM 6.4. *The Elimination Stack is linearizable.*

Proof: Follows from the fact that the above expression is an abstraction of the quotient (See Enea et al. [2023]), with the **bold** actions in the layers being the LPs. ■

Comparison with the Authors' Proof. A proof is given by Hendler et al. [2004] in that paper's Section 5. It is a lengthy proof so, for lack of space, the full review is in in the extended version [Enea et al. 2023] and summarized here. Overall, the correctness argument requires numerous lemmas in the Hendler et al. [2004] proof, mostly focused on establishing a bijection between the active thread and its correspondingly collided passive thread. The authors lay out a few definitions, which are also captured by the quotient. For example, the authors' prose includes:

> *[A] colliding operation op is <u>active</u> if it executes a successful CAS in lines C2 or C7. We say that a colliding operation is <u>passive</u> if op fails in the CAS of line S10 or S19. [underlines added]* – Hendler et al. [2004]

Above the authors' intuitive concept of "active" is captured by the paths in a layer that succeed their CAS, denoted in **bold** in the quotient automaton above. Likewise for "passive" and CAS failure. As mentioned above, the active thread is captured as the bold thread that succeeds its CAS in the bottommost layer; the passive thread is the thread that finds itself collided with in the layers on arcs exiting the bottommost layer.

we show that push and pop operations are paired correctly during collisions. Lemma 5.7. Every passive collider collides with exactly one active collider.

The bottommost layer in the **bold** action, a single push or pop succeeds, colliding with another operation of the oppose type, and passing the element from the push to the pop.

Authors' LPs are given for "active" threads as the time when the second CAS succeeds, and linearization points for "passive" threads "the time of linearization of the matching active-collider operation, and the push colliding-operation is linearized before the pop colliding-operation." The linearization points in the quotient correspond to the bold successful CAS in the bottommost layer in the quotient automaton (this linearizes both a push and a pop). Importantly, every run of the quotient automaton gives a serial linearization order that is a repetition of pairs of active/passive threads. All other executions are equivalent to one such serialized run, upto commutativity.

In summary, as detailed in the extended version [Enea et al. 2023], the quotient naturally and succinctly captures the key concept of the Elimination stack: that a single successful CAS of one type of operation is the LP for that operation as well as the corresponding matched operation. The quotient captures "active" versus "passive" threads (in the automaton layers/states/transitions), as well as this bijection through the runs of the automaton: every run in the automaton contains some number of active/passive pairs and provides a representative serialization order (in each pair the push is serialized before the pop). Linearization points and other logistics of threads preparing/completing are similarly captured by the quotient automaton.

6.4 The Harris et al. Restricted Double-Compare Single-Swap (RDCSS)

RDCSS [Harris et al. 2002] is a restricted version of a double-word CAS which modifies a so-called data address provided that this address and another so-called control address have some given expected values (the tests and the write happen atomically). `RDCSS` attempts a standard CAS on the data address to change the old value into a pointer to a descriptor structure that stores the inputs of the operation. This fails if the data address does not have the expected value. A second standard CAS on the data address is used to write the new value if the control address has the expected value or the old value, otherwise. Faster threads can help complete the operations of slower threads using the information stored in the descriptor.

The traces in the quotient of `RDCSS` interleave successful attempts at modifying the data address with unsuccessful ones. A successful attempt consists of a thread succeeding the first CAS combined with competing threads that fail, followed by another thread succeeding the second CAS (this can be different from the first one in the case of helping) combined with other threads that fail. An unsuccessful attempt may contain just a thread failing the first CAS, or it can contain two successful CASs like a successful attempt (when the data address has the expected value but the control address does not). Proving linearizability of quotient traces is obvious because they make explicit the "evolution" of a data address, oscillating between storing values and descriptors, and which CAS is enabled depending on the value of the control address. See Enea et al. [2023] for more details.

6.5 The Herlihy-Wing Queue

The quotients of some data structures cannot be represented using layer automata. The Herlihy-Wing Queue [Herlihy and Wing 1990] is one such example and it is notorious for linearization points that depend on the future and that *cannot* be associated to fixed statements, see e.g. Schellhorn et al. [2012]! The queue is implemented as an array of slots for items, with a shared variable `back` that indicates the last possibly non-empty slot. An `enq` atomically reads and increments `back` and then later stores a value at that location. A `deq` repeatedly scans the array looking for the first non-empty slot in a doubly-nested loop. We show that the Herlihy-Wing queue quotient can be abstracted by an expression $(\texttt{deqF}^* \cdot (\texttt{enqI})^+ \cdot \texttt{enqW}^* \cdot \texttt{deqT}^*)^*$, where `deqF` captures dequeue scans that need to restart, `deqT` scans succeed, `enqI` reads/increments `back` and `enqW` writes to the slot. For lack of space, a detailed discussion about how this expression abstracts the quotient is given in the extended version [Enea et al. 2023]. Importantly, linearization points in executions represented by this expression are *fixed*, drastically simplifying reasoning from the general case where they are non-fixed.

THEOREM 6.5. *The Herlihy-Wing Queue is linearizable. (see* Enea et al. [2023]*)*

Comparison with the Authors' Proof. Herlihy and Wing [1990] give intuitions of scenarios:

> *Enq execution occurs in two steps, which may be interleaved with steps of other concurrent operations: an array slot is reserved by atomically incrementing back, and the new item is stored in items.* – Sec 4.1 of Herlihy and Wing [1990]

This describes a scenario with unboundedly many threads, though is not yet an argument for why that scenarios is correct. This scenario appears in the quotient as the fact that `enqI` and `enqW` are distinct. To cope with non-fixed LPs (in this and other objects), the authors introduce a proof methodology based on tracking all possible linearizations that could happen in the future. This general methododology complicates the proof. The quotient, by contrast, allows one to consider scenarios along the lines of "one or more enqueuers increment back, possibly some of them write to the array, and then some dequeuers succeed," following the quotient's regular expression. In summary, the quotient here provides the first scenario-based proof of correctness, through representative executions that allow the linearization order to be *fixed* and all other executions are equivalent to one such representative execution up to commutativity.

7 GENERATING CANDIDATE QUOTIENT EXPRESSIONS

In Sec. 6 we showed quotients can be defined for a wide range of concurrent objects, including notoriously difficult ones. We leave the (rather large) question of automated quotient proofs for the general case as future work. Here we take a first step asking, *Can candidate quotient expressions can be generated algorithmically?*

This section answers this question with an algorithm, implementation and experiments showing that, from the source code of concurrent data-structures such as Treiber's stack and the MSQ, candidate quotients expressions (equivalent to those in Sec. 6) can be automatically discovered. We manually confirmed that these generated candidates are indeed sound abstractions of the quotient, a process that can also be automated (perhaps through new forms of induction) in future work.

7.1 Computing Layer Automata

Given a set of layers $\lambda_1,\ldots,\lambda_n$ whose starred union is an abstraction of an object quotient (cf. Theorem 5.6), a layer automaton satisfying Theorem 5.9 can be computed automatically. The algorithm consists of the following steps:

(1) *States*: Compute the automaton abstract states as boolean conjunctions of the weakest preconditions (and their negations) of traces in the *support* of a layer λ_i with $1 \leq i \leq n$. We assume that the initial state can be determined from the object spec.
(2) *Edges*: Whenever a state q implies the precondition of a write layer λ_i with write path k_w, compute every post-state q' that can hold, and add an edge $q \xrightarrow{\lambda_i} q'$. This can be encoded as an assertion violation in a program that assumes q; k_w and asserts the negation of q'.

Table 1. Evaluation of Cion discovering candidate layers from source code.

Example	States $\|Q\|$	# Paths # k_l	# k_w	# Trans. $\|\delta\|$	# Layers $\|\Lambda(O)\|$	Time (s)	# Solver Queries
`evenodd.c`	2	2	2	6	3	52.2	32
`counter.c`	2	3	2	6	5	67.8	36
`descriptor.c`	4	6	2	6	6	160.2	74
`treiber.c`	2	3	2	6	5	71.4	37
`msq.c`	4	9	3	17	7	441.6	314
`listset.c`	7	6	2	59	7	603.8	494

(3) *Self-Loops*: For every state q collect every local layer that is enabled from q and create a single self-loop consisting of a concatenation of all these layers.

7.2 Implementation and Experiments

We built a proof-of-concept implementation of our algorithm, called Cion in ~1,000 lines of OCaml code, using CIL and Ultimate [Heizmann et al. 2018]. Cion is publicly available[6]. We applied Cion to some of the Sec. 6 objects that were amenable to layers. Experiments were run on Ubuntu 22.04 within a Parallels VM on a MacBook Pro M2 with 32GB RAM. Benchmarks are available in Cion repository. We used Ultimate v0.2.1 (`54a68f4`) as a reachability solver, with its default configuration. The results are summarized in Table 1. For each benchmark, we report the number of automaton **States** $|Q|$, the number of local **Paths** $\#k_l$ and number of write paths $\#k_w$. We then report the number of **Trans**itions $|\delta|$ in the automata constructed by Cion and the number of **Layers**, as well as the wall-clock **Time** in seconds, and the number of **Queries** made to the solver (Ultimate). The results show that Cion is able to efficiently generate candidate layer automata for some important and challenging concurrent objects.

8 RELATED WORK

Linearizability proofs. Program logics for compositional reasoning about concurrent programs and data structures have been studied extensively, as mentioned in Sec. 1.1. Improving on the classical Owicki and Gries [1976] and Rely-Guarantee [Jones 1983] logics, numerous extensions of Concurrent Separation Logic [Bornat et al. 2005; Brookes 2004; O'Hearn 2004; Parkinson et al. 2007] have been proposed in order to reason compositionally about different instances of fine-grained concurrency, e.g. [da Rocha Pinto et al. 2014; Dragoi et al. 2013; Jung et al. 2018, 2020; Krishna et al. 2018; Ley-Wild and Nanevski 2013; Nanevski et al. 2019; Raad et al. 2015; Sergey et al. 2015; Turon et al. 2013; Vafeiadis 2008, 2009]. We build on the success of such program logics toward improving the confidence in the correctness of concurrent objects. In the current paper we alternatively focus on the scenario-based reasoning found in the distributed computing literature, and have aimed to capture those scenarios as formally-defined representative executions. In future work it could be interesting to combine the benefits of program logics with those of quotients. Other more distantly related works include: Berdine et al. [2008], Vafeiadis [2010], Bouajjani et al. [2013], Chakraborty et al. [2015], Zhu et al. [2015], and Abdulla et al. [2016].

Reduction. The reduction theory of Lipton [1975] introduced the concept of *movers* to define a program transformation that creates atomic blocks of code. QED [Elmas et al. 2009] expanded Lipton's theory by introducing iterated application of reduction and abstraction over gated atomic actions. CIVL [Hawblitzel et al. 2015] builds upon the foundation of QED, adding invariant reasoning and refinement layers [Kragl and Qadeer 2018; Kragl et al. 2018]. Reasoning via simplifying program

[6]https://github.com/quotientprovers/cion

transformations has also been adopted in the context of mechanized proofs, e.g., [Chajed et al. 2018]. Inductive sequentialization [Kragl et al. 2020] builds upon this prior work, and introduces a new scheme for reasoning inductively over unbounded concurrent executions. The main focus of these works is to define generic proof rules to prove soundness of such program transformations, whose application does however require carefully-crafted artifacts such as abstractions of program code or invariants. Our work takes a different approach and tries to distill common syntactic patterns of concurrent objects into a simpler reduction argument. Our reduction is *not* a form of program transformation since quotient executions are interleavings of actions in the implementation.

9 CONCLUSION

We have shown that scenario-based reasoning about concurrent objects has a formal grounding, answering an open question. The key insight is the concept of a quotient, defined so that it admits *only* representative traces and all other traces are merely equivalent to one of those representatives, up to commutativity. We then gave a language for finitely expressing abstractions of those quotients (as regular or context-free languages) and an inductive and automata-theoretical way of describing them. Our results show that quotients provide a succinct formal foundation for scenario-based reasoning, are capable of capturing a wide range of tricky objects, enhance original authors' correctness arguments, and that discovery of candidate quotient expressions can be automated. In the future will explore further mechanization and other application domains.

DATA-AVAILABILITY STATEMENT

Software that supports Sec. 7 is available on GitHub [Enea et al. 2024] and Zenodo [Koskinen 2024].

ACKNOWLEDGMENTS

We thank Matthew Parkinson and the anonymous reviewers for their feedback on this draft. Koskinen was partially supported by NSF award CCF-2008633 and NSF award CCF-2315363. Enea was partially supported by ANR award SCEPROOF.

REFERENCES

Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh. 2016. Automated Verification of Linearization Policies. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9837)*, Xavier Rival (Ed.). Springer, 61–83. https://doi.org/10.1007/978-3-662-53413-7_4

Josh Berdine, Tal Lev-Ami, Roman Manevich, G. Ramalingam, and Shmuel Sagiv. 2008. Thread Quantification for Concurrent Shape Analysis. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5123)*, Aarti Gupta and Sharad Malik (Eds.). Springer, 399–413. https://doi.org/10.1007/978-3-540-70545-1_37

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. 259–270. https://doi.org/10.1145/1040305.1040327

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2013. Verifying Concurrent Programs against Sequential Specifications. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 290–309. https://doi.org/10.1007/978-3-642-37036-6_17

Stephen Brookes. 2007. A semantics for concurrent separation logic. *Theor. Comput. Sci.* 375, 1-3 (2007), 227–270. https://doi.org/10.1016/j.tcs.2006.12.034

Stephen D. Brookes. 2004. A Semantics for Concurrent Separation Logic. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*. 16–34. https://doi.org/10.1007/978-3-540-28644-8_2

Tej Chajed, M. Frans Kaashoek, Butler W. Lampson, and Nickolai Zeldovich. 2018. Verifying concurrent software using movers in CSPEC. In *OSDI*. https://www.usenix.org/conference/osdi18/presentation/chajed

Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2015. Aspect-oriented linearizability proofs. *Log. Methods Comput. Sci.* 11, 1 (2015). https://doi.org/10.2168/LMCS-11(1:20)2015

Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. 1998. Symmetry Reductions in Model Checking. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1427)*, Alan J. Hu and Moshe Y. Vardi (Eds.). Springer, 147–158. https://doi.org/10.1007/BFb0028741

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings.* 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Alan Martin, Nir Shavit, and Guy L. Steele Jr. 2000. Even Better DCAS-Based Concurrent Deques. In *Distributed Computing, 14th International Conference, DISC 2000, Toledo, Spain, October 4-6, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1914)*, Maurice Herlihy (Ed.). Springer, 59–73. https://doi.org/10.1007/3-540-40026-5_4

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 287–300. https://doi.org/10.1145/2429069.2429104

Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. 2009. Deny-Guarantee Reasoning. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 363–377. https://doi.org/10.1007/978-3-642-00590-9_26

Simon Doherty, David Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul Alan Martin, Mark Moir, Nir Shavit, and Guy L. Steele Jr. 2004. DCAS is not a silver bullet for nonblocking algorithm design. In *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*, Phillip B. Gibbons and Micah Adler (Eds.). ACM, 216–224. https://doi.org/10.1145/1007912.1007945

Cezara Dragoi, Ashutosh Gupta, and Thomas A. Henzinger. 2013. Automatic Linearizability Proofs of Concurrent Objects with Cooperating Updates. In *CAV '13 (LNCS, Vol. 8044)*. Springer, 174–190.

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2009. A calculus of atomic actions. In *POPL*. https://doi.org/10.1145/1480881.1480885

Constantin Enea, Parisa Fathololumi, and Eric Koskinen. 2023. Scenario-Based Proofs for Concurrent Objects [Extended Version]. arXiv:2301.05740 [cs.PL]

Constantin Enea, Parisa Fathololumi, and Eric Koskinen. 2024. CION: Concurrent Trace Reductions. https://github.com/quotientprovers/cion

Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* Lecture Notes in Computer Science, Vol. 1032. Springer. https://doi.org/10.1007/3-540-60761-7

Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings (Lecture Notes in Computer Science, Vol. 2508)*, Dahlia Malkhi (Ed.). Springer, 265–279. https://doi.org/10.1007/3-540-36108-1_18

Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. Automated and Modular Refinement Reasoning for Concurrent Programs. In *CAV*. https://doi.org/10.1007/978-3-319-21668-3_26

Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. 2018. Ultimate Automizer and the Search for Perfect Interpolants - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10806)*, Dirk Beyer and Marieke Huisman (Eds.). Springer, 447–451. https://doi.org/10.1007/978-3-319-89963-3_30

Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A scalable lock-free stack algorithm. In *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*, Phillip B. Gibbons and Micah Adler (Eds.). ACM, 206–215. https://doi.org/10.1145/1007912.1007944

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

Cliff B. Jones. 1983. Specification and Design of (Parallel) Programs. In *IFIP Congress*. 321–332.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 637–650. https://doi.org/10.1145/2676726.2676980

Eric Koskinen. 2024. *quotientprovers/cion: oopsla2024-artifact: Version used in evaluation for the OOPSLA'24 paper "Scenario-Based Proofs for Concurrent Objects"*. https://doi.org/10.5281/zenodo.10814650

Dexter Kozen. 1997. Kleene Algebra with Tests. *ACM Trans. Program. Lang. Syst.* 19, 3 (1997), 427–443. https://doi.org/10.1145/256167.256195

Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. 2020. Inductive sequentialization of asynchronous programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 227–242. https://doi.org/10.1145/3385412.3385980

Bernhard Kragl and Shaz Qadeer. 2018. Layered Concurrent Programs. In *CAV*. https://doi.org/10.1007/978-3-319-96145-3_5

Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. 2018. Synchronizing the Asynchronous. In *CONCUR*. https://doi.org/10.4230/LIPIcs.CONCUR.2018.21

Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. 2018. Go with the flow: compositional abstractions for concurrent data structures. *PACMPL* 2, POPL (2018), 37:1–37:31. https://doi.org/10.1145/3158125

Ruy Ley-Wild and Aleksandar Nanevski. 2013. Subjective auxiliary state for coarse-grained concurrency. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. 561–574. https://doi.org/10.1145/2429069.2429134

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975). https://doi.org/10.1145/361227.361234

Antoni W. Mazurkiewicz. 1986. Trace Theory. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986 (Lecture Notes in Computer Science, Vol. 255)*, Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg (Eds.). Springer, 279–324. https://doi.org/10.1007/3-540-17906-2_30

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC '96*. ACM, 267–275.

Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. 2019. Specifying concurrent programs in separation logic: morphisms and simulations. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 161:1–161:30. https://doi.org/10.1145/3360587

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*. 49–67. https://doi.org/10.1007/978-3-540-28644-8_4

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007). https://doi.org/10.1016/j.tcs.2006.12.035

Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Verifying linearizability with hindsight. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, Andréa W. Richa and Rachid Guerraoui (Eds.). ACM, 85–94. https://doi.org/10.1145/1835698.1835722

Susan S. Owicki and David Gries. 1976. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM* 19, 5 (1976), 279–285. https://doi.org/10.1145/360051.360224

Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. 2007. Modular verification of a non-blocking stack. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. 297–302. https://doi.org/10.1145/1190216.1190261

Azalea Raad, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent Local Subjective Logic. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 710–735. https://doi.org/10.1007/978-3-662-46669-8_29

Gerhard Schellhorn, Heike Wehrheim, and John Derrick. 2012. How to Prove Algorithms Linearisable. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 243–259.

William N Scherer III, Doug Lea, and Michael L Scott. 2006. Scalable synchronous queues. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. 147–156.

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 333–358. https://doi.org/10.1007/978-3-662-46669-8_14

R. K. Treiber. 1986. *Systems Programming: Coping with Parallelism*. Technical Report RJ 5118. IBM Almaden Research Center.

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 377–390. https://doi.org/10.1145/2500365.2500600

V. Vafeiadis. 2008. *Modular fine-grained concurrency verification*. Ph. D. Dissertation. University of Cambridge.

Viktor Vafeiadis. 2009. Shape-Value Abstraction for Verifying Linearizability. In *VMCAI '09: Proc. 10th Intl. Conf. on Verification, Model Checking, and Abstract Interpretation (LNCS, Vol. 5403)*. Springer, 335–348.

Viktor Vafeiadis. 2010. Automatically Proving Linearizability. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6174)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.). Springer, 450–464. https://doi.org/10.1007/978-3-642-14295-6_40

Viktor Vafeiadis and Matthew J. Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4703)*, Luís Caires and Vasco Thudichum Vasconcelos (Eds.). Springer, 256–271. https://doi.org/10.1007/978-3-540-74407-8_18

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2015. Poling: SMT Aided Linearizability Proofs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. 3–19.

Received 21-OCT-2023; accepted 2024-02-24