Scenario-based Proofs for Concurrent Objects

CONSTANTIN ENEA, École Polytechnique, France
ERIC KOSKINEN, Stevens Institute of Technology, US

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.

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 2008a]. 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 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 [1996a] 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 [2008b]). 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 [**?**] where later determined to contain bugs in unconsidered scenarios [**?**].

Authors' addresses: Constantin Enea, École Polytechnique, France; Eric Koskinen, Stevens Institute of Technology, US.

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 [O'Hearn 2007; **?**], RGSep [**?**], Deny-Guarantee [**?**], Views [**?**], Iris [Jung et al. 2018, 2015] and interactive proof tools such as Iris.

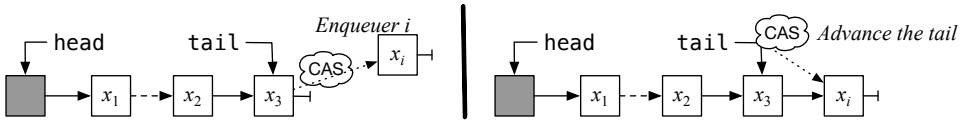
The goal of this paper is orthogonal and focuses on finding a formal grounding for the operational, scenario-based correctness arguments present in the algorithms literature. To this end, our methodology is based on taking representative interleaved traces upfront and using commutativity-based equivalence 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] 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.

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 unlink x_1 with a CAS operation, 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 2008b] 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 $r_{\texttt{next}}$:

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

where T is the (unbounded) set of all threads excluding τ_{enq}. Above $r_{\texttt{next}}$ 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 $r_{\texttt{tail}}$ and $r_{\texttt{head}}$ 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 $(r_{\texttt{next}} + r_{\texttt{tail}} + r_{\texttt{head}})^*$. (Below we will further refine this approximation with stateful automata.)

This 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 $r_{\texttt{next}}$, $r_{\texttt{tail}}$ and $r_{\texttt{head}}$ 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

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

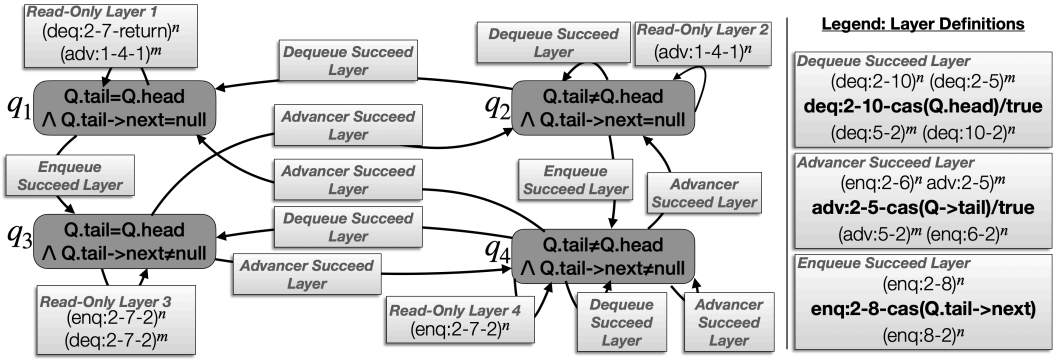


Fig. 1. Layer automaton for the Michael/Scott Queue.

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 **?**, 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.

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 [1996b] queue, the Scherer III et al. [2006] synchronous reservation queue, the Herlihy and Wing [1990] queue, the **?** elimination stack, and the Restricted Double-Compare Single-Swap (RDCSS) [**?**]. 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. We plan to release Cion on GitHub. Benchmark sources and the tool output are in the supplementary materials.

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.

```
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.

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 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.

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).

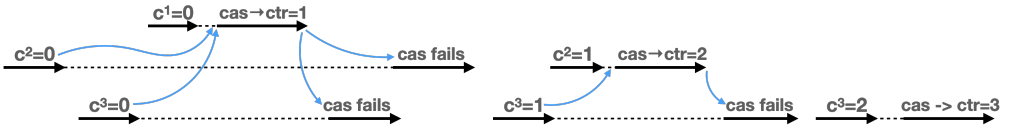


Fig. 3. The steps of an execution with three increment-only threads whose actions are aligned horizontally. For readability, we rename the local variable c in thread i to c^i. The curved blue arrows depict data-flow dependencies between reads/writes of `ctr`.

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 $BExp ::= b \in \mathsf{B} \mid b \cdot b \mid b + b \mid \overline{b} \mid 0 \mid 1$, 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 $k_1 + k_2 = k_2$, and we assume $\mathcal{K}$ is *-continuous [Kozen 1990].

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

Atomic read-write (ARW). We conservatively extend KAT with a syntactic notation $\langle \mathsf{b} \cdot \mathsf{a} \rangle$, 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 "$\langle\!\langle$" and "$\rangle\!\rangle$" whose semantics are the identity relation. For example a compare-and-swap `cas(x,v,v')` can be represented as $(\langle\!\langle \mathtt{[x=v]} \cdot \mathtt{x:=v'} \rangle\!\rangle \cdot k) + (\overline{\mathtt{[x=v]}} \cdot k')$, 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.,* $\mathtt{x} := args_i$. 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 $Meth(O)$.

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

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

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

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 Apx. A. 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 $\mathsf{call}\ m(\vec{v})$ 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 $\rightsquigarrow$: $C \times (\mathcal{T} \times \mathcal{L}) \times C$, labeled with a thread id and a label.

An object O is acted on by a finite ***environment*** $\mathcal{E} : \mathcal{T} \to O \times \vec{Val}$, specifying which threads invoke which methods, with which argument values. Val denotes a set of values and $\vec{Val}$ 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 $C_0 \rightsquigarrow \ldots \rightsquigarrow C_n$ 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 $t \in \mathrm{dom}(T^f)$. 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*** $\tau \in Traces$ 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 $m \in Meth(O)$ 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 $lp(\tau) : \mathcal{T} \to \mathbb{N}$ 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 [?] composed with a symmetry reduction [?] 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 $\equiv_1$):

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

assuming that $\mathtt{ctr} > 0$ at configuration C_1 (recall that $\overline{[\mathtt{c}_t=\mathtt{ctr}]}$ 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, $C_0 \xrightarrow{(t:a)} C_1 \xrightarrow{(t':b)} C_2$ and $C_0 \xrightarrow{(t':a)} C_1 \xrightarrow{(t:b)} C_2$ are equivalent up to thread renaming.

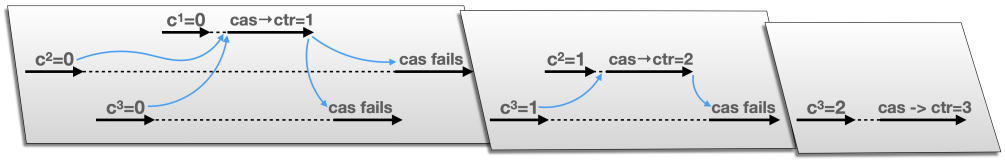
We define a quotient of an object as a subset of its traces that is *complete* in the sense that it represents every other trace up to commutative reorderings or thread renaming, and that is *optimal* in that sense that it does not contain two traces that are equivalent up to commutativity. Optimality 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:

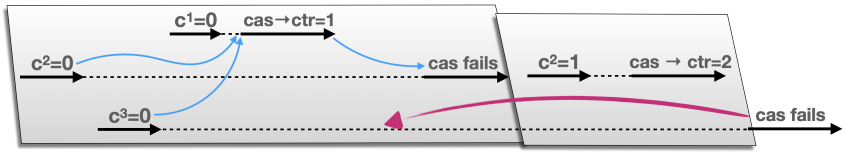
- $\forall \tau \in [\![O]\!]. \exists \tau', \tau''. \tau \simeq \tau' \wedge \tau' \equiv \tau'' \wedge \tau'' \in \langle\!\lfloor O \rfloor\!\rangle$ (completeness), and
- $\forall \tau, \tau' \in \langle\!\lfloor O \rfloor\!\rangle. \ \tau \not\equiv \tau'$ (optimality)

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

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. A linearization point mapping $lp(\tau)$ of a trace τ is ***robust against reorderings*** if for every two threads t_1 and t_2, if the linearization points of t_1 and t_2 are possibly commutative labels, then the operations of t_1 and t_2 are S-commutative.

Theorem 3.5. *Let $\tau \equiv \tau'$ be two equivalent traces. If τ is linearizable w.r.t. some specification S via a linearization point mapping $lp(\tau)$ that is robust against 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 of O, provided that the used linearization point mappings are robust against reorderings (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 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,

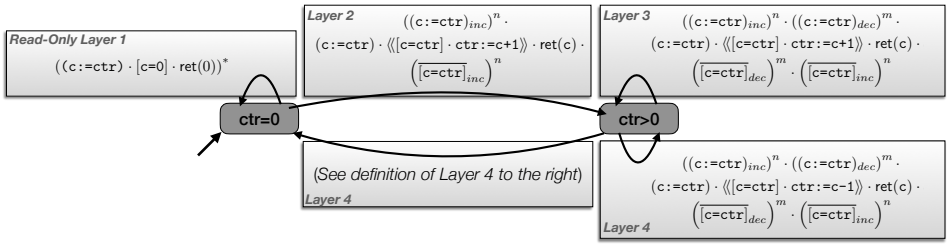


Fig. 4. An expression representing a quotient of the Counter. For readability we present it as four sub-expressions called "layers" whose composition with regular expression operators (concatenation, union, star) is represented using an automaton (all states are accepting). We subscript the primitives 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.

- $[\![\omega]\!] = \{t : \omega \mid t \in \mathcal{T}\}$, 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
- $[\![\mathsf{expr}_1 \cdot \mathsf{expr}_2]\!] = [\![\mathsf{expr}_1]\!], [\![\mathsf{expr}_2]\!]$, concatenation of interpretations

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

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

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

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):

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

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

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[3] 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 $[\![a]\!] : (\Sigma_{lo} \times \Sigma_{gl}) \rightarrow (\Sigma_{lo} \times \Sigma_{gl})$ and $[\![b]\!] : (\Sigma_{lo} \times \Sigma_{gl}) \rightarrow \{true, false\}$ 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:

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

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.

[3]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.

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

- *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.

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 $k_w = (\texttt{c:=ctr}) \cdot \langle\!\langle [\texttt{c=ctr}] \cdot \texttt{ctr:=c+1} \rangle\!\rangle \cdot \text{ret}(\texttt{c})$, $supp(\text{Layer 3}) = \{(\texttt{c:=ctr})_{inc} \cdot k_w \cdot \overline{[\texttt{c=ctr}]}_{inc}, (\texttt{c:=ctr})_{dec} \cdot k_w \cdot \overline{[\texttt{c=ctr}]}_{dec}\}$. 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 Apx. B.

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* $\mathsf{expr} = (\lambda_1 + \ldots + \lambda_n)^*$ *is an abstraction of a quotient of O if*

- *the layers cover all statements in the implementation:* $\Pi(\mathsf{expr}) \subseteq \Pi(O)$ *and for each primitive action, test or ARW k_p in k_{m_i} for some i, there exists a path in* $\Pi(\mathsf{expr})$ *which contains k_p,*
- *for every path* $k \in \Pi(\mathsf{expr})$ *and every execution ρ of O starting in a reachable configuration that represents*[4] *an interleaving $k \mid\mid \vec{k}_w$, where $\vec{k}_w$ is a sequence of write paths in* $\Pi(\mathsf{expr})$*,*
  - *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 $\texttt{(c:=ctr)} \cdot \langle\!\langle \texttt{[c=ctr]} \cdot \texttt{ctr:=c+1} \rangle\!\rangle \cdot \texttt{ret(c)}$. 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 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).

Layer Automata. The "simple" starred union composition of layers in Theorem 5.6 can be refined further using standard reachability analyses. For instance, as shown in Figure 4 for the Counter, the read-only "decrement returning 0" layer cannot occur after one successful increment layer. We represent such constraints on the order in which layers can occur using automata. Another example of such an automaton was seen for the Michael-Scott queue in Fig. 1 in Sec. 1. A formalization of these

[4] An execution ρ represents an interleaving $k \mid\mid \vec{k}_w$ 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).

layer automata can be found in Apx. C. 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). Sec. 7 presents a prototype implementation capable of generating candidate layer quotients, represented as layer automata.

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 [1996a] 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	Apx. I	simple cas loop
? stack	**Sec. 6.3**	elimination, submodule, LP helping
? 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	Apx. L	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 many in this section (with bold **Sec 6._** in the **Quotient** column), with further detail in Apx. G–N.

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.)

```
1 int enq(int v){ loop {
2  node_t *node=...;
3  node->val=v;
4  tail=Q.tail;
5  next=tail->next;
6  if (Q.tail==tail) {
7   if (next==null) {
8    if (CAS(&tail->next,
9            next,node))
10      ret 1;
11 } } } }
```

```
1 int deq(){ loop {
2  int pval;
3  head=Q.head;tail=Q.tail;
4  next=head->next;
5  if (Q.head==head) {
6   if (head==tail) {
7    if (next==null) ret 0;
8   } else {
9    pval=next->val;
10   if (CAS(&Q->head,
11           head,next))
12     ret pval;
13  } } } } }
```

Factored out tail advancement: (see notes below)

```
1 adv(){ loop {
2  tail=Q.tail;
3  next=tail->next;
4  if (next!=null){
5   if (CAS(&Q->tail,
6     tail,next))
7    ret 0;
8  }
9 } }
```

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.

The layer automaton that abstracts a quotient of MSQ is shown in Fig. 1 (details in Apx. G). The states track whether `Q.tail=Q.head` and whether `Q.tail->next=null`, in rounded dark boxes. Edges are labeled with layers, defined to the right in Fig. 1. These three layers 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.

Theorem 6.1. *The Michael-Scott Queue is linearizable.*

Proof: Linearization points (LPs) are the successful CAS operations in the *{Dequeue,Advancer,Enqueue} Succeed Layer*s (also in **bold** in the Fig. 1 layer definitions), as well as the the first (or any) action in the Read-Only layers. Per Thm. G.1, the quotient expression (layer automaton) is an abstraction of the quotient and thus we have given LPs for all executions of the MSQ.

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

Proof Element	Herlihy and Shavit [2008b]	**Quotient Proof**
ADT states	"queue is nonempty," "tail is lagged"	ADT states, e.g. (`Q.tail=Q.head` $\wedge$ `Q.tail->next` $\neq$ `null`)
Concurrent threads	"some other thread"	Superscripting $(\ldots)^n$
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 [2008b] 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.

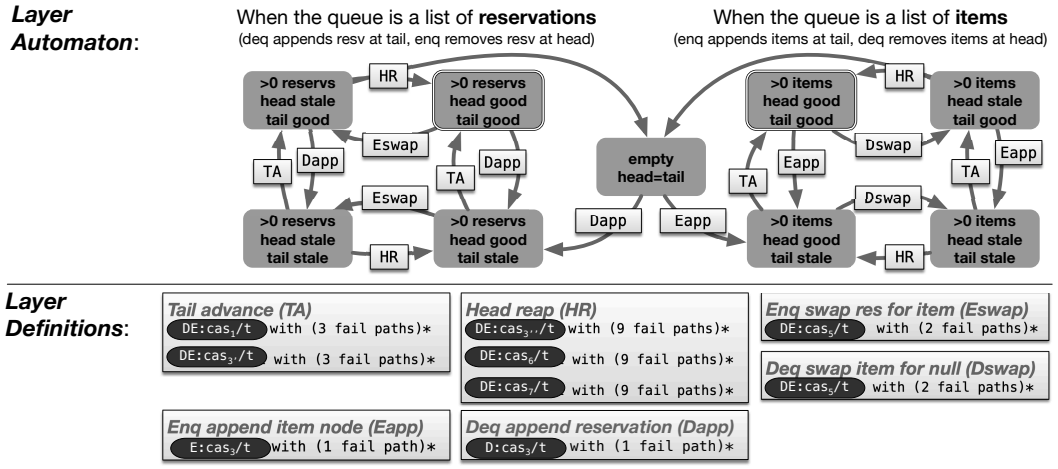


Fig. 5. Layer automaton for the synchronous SLS queue. Layers' acronyms and their definitions are given in the lower half of the figure. For conciseness, layer definitions do not split the prefix/suffix of the read paths.

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 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 Apx. D, 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 6.2. *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 meet the sequential spec. of a queue. (Detail in Apx. D, Thm. H.1.)

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 Apx. D. For lack of space, some discussion of the authors' quotes can be found in Apx. N.4.

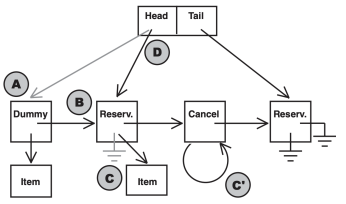
The authors split the enqueue operation into two linearization points: a "reservation linearization point" and a later "follow up linearization point," so that synchronous, blocking enqueue implementations are a single reservation LP and then repeated follow-up LPs (as if the client is repeatedly checking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

> *[Regarding enqueue,] the reservation linearization point for this code path occurs at line 10 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.

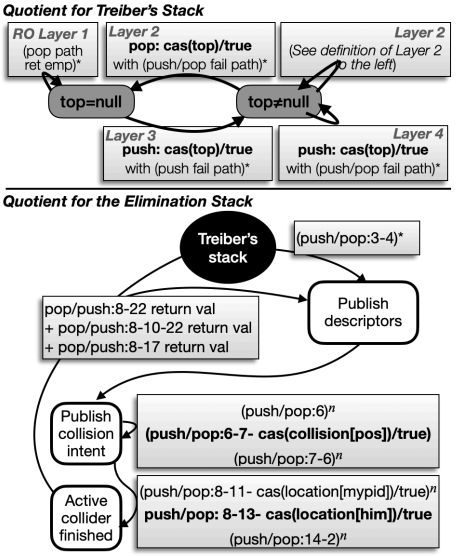
> *The other case occurs when the queue consists of reservations (requests for data), and is depicted [to the right]. In this case, after originally reading the head node (step A), we read its successor (line 21/step B) and verify consistency (line 22). Then, we attempt to supply our data to the head-most reservation (line 25/C). If this succeeds, we dequeue the former dummy node (26/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

```
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



(b) Stack Quotients

Fig. 6. Elimination Stack

captured directly in the ADT states and the HR layers' transitions. The authors' prose also discusses failure paths (see Apx. N.4) and retry, which are also captured in the layer definitions.

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 **?** 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 Apx. I. 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 Apx. I for more detail, as well as Lemma I.1 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).

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 6.3. *The Elimination Stack is linearizable.*

Proof: Follows from the fact that the above expression is an abstraction of the quotient (Thm. J.1), with the **bold** actions in the layers being the LPs.

Comparison with the authors' proof. A proof is given by **?** in that paper's Section 5. It is a lengthy proof so, for lack of space, the full review is in Apx. N.2 and summarized here. Overall, the correctness argument requires numerous lemmas in the **?** 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 active if it executes a successful CAS in lines C2 or C7. We say that a colliding operation is passive if op fails in the CAS of line S10 or S19. [underlines added]* – **?**

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 Apx. N.2, 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 [**?**] 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 App. K 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 *can not* 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

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

Example	States $\|Q\|$	# Paths # k_l	# Paths # k_w	# Trans. $\|\delta\|$	# Layers $\|\Lambda(O)\|$	Time (s)	# Solver Queries
evenodd.c	2	2	2	6	3	50.8	32
counter.c	2	3	2	6	4	63.3	36
descriptor.c	4	6	2	6	5	155.2	74
treiber.c	2	3	2	6	4	70.3	37
msq.c	4	9	3	17	7	437.6	314
listset.c	7	6	2	77	8	466.9	532

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 Apx. M. Importantly, linearization points in executions represented by this expression are *fixed*, drastically simplifying reasoning from the general case where they are non-fixed.

Theorem 6.4. *The Herlihy-Wing Queue is linearizable. (see Thm. M.2)*

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.

The algorithm exploits our reduction to two-thread reasoning and automaton representation of layer quotients (Apx. C). The algorithm is in Apx. C.2 but, briefly, involves (i) computing automaton states using weakest preconditions, (ii) computing the possible post-states of write paths k_w, and which local paths are feasible interleavings with those write paths (exploiting pair-wise reasoning about paths), and (iii) computing which automaton self-loops are possible via local-only layers.

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 will soon be released publicly on

GitHub, with an artifact of the experiments also available. We applied Cion[5] to some of the Sec. 6 objects that were amenable to layers. Benchmarks are available in the supplemental materials. The results are summarized in Table 1 and the Cion output is in `tooloutput.pdf` in the supplement. 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: **?**, **?**, **?**, **?**, **?**, and **?**.

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 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. 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.

[5] Run on Ubuntu 18, Parallels, Macbook Pro M1, 16GB RAM.

REFERENCES

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetkAT: semantic foundations for networks. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 113–126. https://doi.org/10.1145/2535838.2535862

Timos Antonopoulos, Eric Koskinen, and Ton Chanh Le. 2019. Specification and inference of trace refinement relations. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 178:1–178:30. https://doi.org/10.1145/3360604

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

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

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

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.

Loris D'Antoni and Margus Veanes. 2017. The power of symbolic automata and transducers. In *International Conference on Computer Aided Verification*. Springer, 47–67.

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

Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. 2018. Order out of Chaos: Proving Linearizability Using Local Views. In *DISC 2018*.

Yotam M. Y. Feldman, Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. 2020. Proving highly-concurrent traversals correct. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 128:1–128:29. https://doi.org/10.1145/3428196

Michael Greenberg, Ryan Beckett, and Eric Hayden Campbell. 2022. Kleene algebra modulo theories: a framework for concrete KATs. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 594–608. https://doi.org/10.1145/3519939.3523722

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

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. 2002. Temporal-Safety Proofs for Systems Code. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*. 526–538.

Maurice Herlihy and Nir Shavit. 2008a. *The art of multiprocessor programming*. Morgan Kaufmann.

Maurice Herlihy and Nir Shavit. 2008b. *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

Dexter Kozen. 1990. On Kleene Algebras and Closed Semirings. In *Mathematical Foundations of Computer Science 1990, MFCS'90, Banská Bystrica, Czechoslovakia, August 27-31, 1990, Proceedings (Lecture Notes in Computer Science, Vol. 452)*, Branislav Rovan (Ed.). Springer, 26–47. https://doi.org/10.1007/BFb0029594

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

M.M. Michael and M.L. Scott. 1996a. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC*.

Maged M. Michael. 2004. *ABA Prevention Using Single-Word Instructions*. Technical Report RC 23089. IBM Thomas J. Watson Research Center.

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

Mark Moir and Nir Shavit. 2004. Concurrent Data Structures. In *Handbook of Data Structures and Applications.*, Dinesh P. Mehta and Sartaj Sahni (Eds.). Chapman and Hall/CRC. https://doi.org/10.1201/9781420035179.ch47

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

Damien Pous. 2015. Symbolic algorithms for language equivalence and Kleene algebra with tests. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 357–368.

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.

Scenario-based Proofs for Concurrent Objects
Appendix

A UNABRIDGED CONCURRENT OBJECT SEMANTICS

We define an operational semantics for concurrent objects as sets of executions that interleave steps of a number of method invocations executed by different threads. For simplicity, we assume that every thread invokes a single method, which is without loss of generality provided that thread ids are modeled as additional inputs. Method implementations are assumed to be given as KAT expressions, as described in Section 2.

Client environments. An object O is acted on by a finite ***environment*** $\mathcal{E} : \mathcal{T} \to O \times \overline{Val}$, specifying which threads invoke which methods, with which argument values. We use $\mathcal{T}$ to denote the set of thread ids, and Val an unspecified set of values ($\overline{Val}$ denotes the set of tuples of values). We assume that $\mathcal{T}$ is equipped with a total order $<$ that will be used to define representatives of equivalence classes up to symmetry (renaming of thread ids).

States. We assume that each test or action in a method implementation acts on a local state, whose content can be accessed only by the thread executing that test/action, and possibly a shared state which can be read or modified by any thread in the environment. As expected, we assume that each local state contains a valuation for the arguments of an invocation $\vec{x}$ and, once a thread has finished its execution, its local state contains the return values $\vec{v}$. A precise formalization of local/shared states is irrelevant to our development and we omit it for readability. Let Σ_{lo} and Σ_{gl} denote the set of local and shared states, respectively.

A thread executes an implementation given by a KAT expression k, according to the rules below. We assume that semantics of tests $[\![b]\!] : (\Sigma_{lo} \times \Sigma_{gl}) \to \mathbb{B}$ and actions $[\![a]\!] : (\Sigma_{lo} \times \Sigma_{gl}) \to (\Sigma_{lo} \times \Sigma_{gl})$ is provided (or generated from the language/program).

Non-deterministic single-thread execution. Given an environment $\mathcal{E}$, a step of a single thread t is a relation on $\Sigma_{lo} \times \Sigma_{gl} \times (\mathcal{K} \cup \{\bot\})$ where $\bot$ indicates that a thread has completed. We denote this relation as $\sigma_l, \sigma_g, k \downarrow_\ell \sigma_l', \sigma_g', k'$, which optionally involve label ℓ. ***Labels*** are taken from the set of possible labels $\mathcal{L} \subseteq \mathsf{A} \cup \mathsf{B} \cup \mathsf{call}\ m(\vec{v}) \cup \mathsf{ret}(\vec{v}) \cup \langle\!\langle b \cdot a \rangle\!\rangle$ which includes primitive actions, primitive tests, invocations, returns or ARWs. (We here write call $m(\vec{v})$ with free variables to refer to the set of all invocations and similar for returns and ARWs.) The labeled single-step semantics are now defined inductively on k as follows:

$$\frac{\mathcal{E}(t) = (m(\vec{x})/\vec{v} : k_m, \vec{v})}{\sigma_l^0, \sigma_g, \epsilon \downarrow_{\mathsf{call}\ m(\vec{v})} \sigma_l^0[args_i \mapsto v_i], \sigma_g, k_m} \qquad \frac{\vec{v} = \sigma_l(\vec{v})}{\sigma_l, \sigma_g, \mathsf{ret}(\vec{v}) \downarrow_{\mathsf{ret}(\vec{v})} \sigma_l, \sigma_g, \bot}$$

$$\frac{}{\sigma_l, \sigma_g, k + k' \downarrow \sigma_l, \sigma_g, k} \qquad \frac{}{\sigma_l, \sigma_g, k + k' \downarrow \sigma_l, \sigma_g, k'} \qquad \frac{\sigma_l, \sigma_g, k \downarrow \sigma_l', \sigma_g', 1}{\sigma_l, \sigma_g, k \cdot k' \downarrow \sigma_l', \sigma_g', k'}$$

$$\frac{}{\sigma_l, \sigma_g, k^* \downarrow \sigma_l, \sigma_g, k \cdot k^*} \qquad \frac{}{\sigma_l, \sigma_g, k^* \downarrow \sigma_l, \sigma_g, 1}$$

$$\frac{a \neq \langle \quad (\sigma_l', \sigma_g') = [\![a]\!](\sigma_l, \sigma_g)}{\sigma_l, \sigma_g, a \downarrow_a \sigma_l', \sigma_g', 1} \qquad \frac{[\![b]\!](\sigma_l, \sigma_g) = \mathsf{true}}{\sigma_l, \sigma_g, b \downarrow_b \sigma_l, \sigma_g, 1}$$

$$\frac{[\![b]\!](\sigma_l, \sigma_g) \quad (\sigma_l', \sigma_g') = [\![a]\!](\sigma_l, \sigma_g)}{\sigma_l, \sigma_g, \langle\!\langle b \cdot a \rangle\!\rangle \downarrow_{\langle\!\langle b \cdot a \rangle\!\rangle} \sigma_l', \sigma_g', 1}$$

The first rule is for invocation, assuming that the environment for this thread specifies that m should be invoked with arguments $\vec{v}$. These arguments are recorded in the local state, and an invocation

label is generated; σ_l^0 is a fixed initial local state. The second rule applies when execution reaches a return statement and a label is generated with the values provided in the local state variables $\vec{v}$. Note that invocation/return labels are not invocation/return actions because they contain *values* rather than arguments. The subsequent 5 rules are built atop the standard non-deterministic semantics of KAT expressions, without any labels being generated. The last 3 rules are for atomic actions a, atomic tests b, and ARWs $\langle b \cdot a \rangle$, with the respective labels generated. When a test b does not hold, the successor is undefined (and similar when atomic test b in $\langle b \cdot a \rangle$ does not hold). We further define $\sigma_l, \sigma_g, k \Downarrow_\ell \sigma_l^n, \sigma_g^n, k^n$, relating triples from a sequence $(\sigma_l, \sigma_g, k) \downarrow (\sigma_l^1, \sigma_g^1, k^1) \downarrow \cdots \downarrow_\ell (\sigma_l^n, \sigma_g^n, k^n)$ where only the final $\downarrow$ transition produces a label. (*i.e.,* the intermediate label-free nondeterminism has been resolved.)

The rules above give a semantics to steps of a thread assuming a certain shared state σ_g, and can be extended to sequences of steps assuming that the shared state can be changed arbitrarily in between every two steps. Formally, given a KAT expression k, an ***execution of*** k starting from a local state σ_l and global state σ_g is defined as a sequence of triples $\sigma_l^i, \sigma_g^i, k^i$ with $0 \leq i \leq n$ such that: (1) $\sigma_l^0 = \sigma_l$, $\sigma_g^0 = \sigma_g$, $k^0 = k$, (2) $\sigma_l^i, \sigma_g^i, k^i \Downarrow_\ell \sigma_l^{i+1}, \sigma_g^{i+1}, k^{i+1}$ for all i even, (3) $\sigma_l^i = \sigma_l^{i+1}$ and $k^i = k^{i+1}$ for all i odd, and (4) $k^n = 1$. Note that σ_g is unconstrained in (3).

Example A.1. Consider k_{inc} as defined in Example 2.2 and $\sigma_l^0 = [\texttt{c} \mapsto \texttt{undef}]$ and $\sigma_g^0 = [\texttt{ctr} \mapsto 0]$. Here is one execution of k_{inc}:

Step 0 :	$(\sigma_l^0, \sigma_g^0, (\texttt{c:=ctr} \cdots)^*)$,	
Step 1 :	$(\sigma_l^0, \sigma_g^0, (\texttt{c:=ctr} \cdots) \cdot (\texttt{c:=ctr} \cdots)^*)$,	Unfold * via $\downarrow$
Step 2 :	$(\sigma_l^0, \sigma_g', (\texttt{c:=ctr} \cdots) \cdot (\texttt{c:=ctr} \cdots)^*)$,	Global state changed arbitrarily
Step 3 :	$(\sigma_l^0[\texttt{c} \mapsto \sigma_g'(\texttt{ctr})], \sigma_g', 1 \cdot (\texttt{c:=ctr} \cdots)^*), \ldots$	Reduce an action via $\downarrow_{\texttt{c:=ctr}}$

A.1 Executions, Traces, Linearizability

The set of executions of a concurrent object O in the context of an environment $\mathcal{E}$ are defined as interleavings of single-thread executions, acting on the shared state and their local states, with nondeterministic scheduling.

A configuration $C \in (\sigma_g, T)$ 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. The initial configuration is defined by $C_0 = (\sigma_g^0, \emptyset)$ where σ_g^0 is a fixed initial shared state. Let $\mathcal{C}$ denote the set of configurations.

An ***execution*** of O is a sequence of configurations and labeled transitions over the threads specified by an environment $\mathcal{E}$. The transition relation $\Rightarrow : \mathcal{C} \times (\mathcal{T} \times \mathcal{L}) \times \mathcal{C}$ is defined as:

$$\frac{\mathcal{E}(t) = (m(\vec{x})/\vec{v} : k_m, \vec{v}) \quad T(t) \text{ undefined}}{(\sigma_g, T) \xrightarrow{(t:\text{call } m(\vec{v}))} (\sigma_g', T[t \mapsto (\sigma_l^0[args_i \mapsto v_i], k_m)])}$$

$$\frac{T(t) = (\sigma_l, k) \quad \sigma_l, \sigma_g, k \Downarrow_\ell \sigma_l', \sigma_g', k'}{(\sigma_g, T) \xrightarrow{(t:\ell)} (\sigma_g', T[t \mapsto (\sigma_l', k')])}$$

A transition $\rightarrow$ is possible for any thread whose k is not $\bot$. The first rule models a new thread invoking a method according to the environment. In the second, the thread t takes a $\Downarrow_\ell$ step, producing label ℓ, and the configuration is updated with the new global state and the new (σ_l', k') for thread t.

B PROOF OF THEOREM 5.6

We reason by induction on the number of paths in $\Pi(\text{expr})$ in a completed execution ρ of O that are either (1) write paths but they are interleaved with actions of other threads, or (2) local paths

but they are interleaved with more than two write paths in $\Pi(\text{expr})$, or with a single write path in $\Pi(\text{expr})$ but together with this path, it does not form a support of a layer in expr.

The base case of the induction is trivial: since every label of a transition in ρ belongs to some path in $\Pi(\text{expr})$, and all paths are interleaved as prescribed by the layers, then clearly, the trace of ρ is in the interpretation of expr.

For the induction step, consider first a write path that is interleaved with actions of other threads. Let ρ' be the minimal subsequence of ρ that contains only steps of that path and all the other write paths that interleave with it. By the induction hypothesis, the latter write paths execute without interruption. This execution is feasible starting from the first configuration of ρ', because we removed only local actions that do not affect enabled-ness of other concurrently executing steps. Applying the WPC condition, there exists an execution ρ'' strongly equivalent to ρ' where all paths execute without interruption. Since ρ'' passes through the same sequence of shared states (modulo stuttering) it can "replace" ρ' in ρ. The trace of the obtained execution is a sequence of layers which ends the proof.

Second, consider a local path that is interleaved with more than two write paths. Similarly to the previous case, one can extract only the steps of that path and all the other write paths with which it interleaves, apply the LPC condition to produce an equivalent sub-execution where that path interleaves with at most one other write path, and then, re-insert the obtained sub-execution into the original execution.

C LAYER AUTOMATA

C.1 Automaton Representation of Layer Quotients

We now show that layer quotients can be represented as automata, as mentioned at the end of Sec. 5. These *layer automata* are a convenient representation of the quotient and, as shown in Sec. 7, can be automatically derived from source code. In general, objects can reach unboundedly many configurations and different layers are enabled/disabled from different configurations, *e.g.*, the layer λ_{dec0} of O_{ctr} in Example 5.3 is enabled only when `ctr` is 0. 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. We therefore describe a more convenient representation as a *layer automaton*, in which the states represent abstractions (sets) of concrete configurations in executions (as defined in Section 2) and the transitions are labeled by basic layer expressions.

Definition C.1 (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 C.2. *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 C.3. *(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* $[\![$expr$]\!]$ *that is robust against re-ordering, then the object is linearizable.*

C.2 Computing Layer Automata

In Sec. 7 we discuss how candidate layer automata can be computed for some canonical examples. This section explains the algorithm in detail. 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 C.2 can be computed automatically. For lack of space we only sketch the procedure. The algorithm consists of the following steps:

(1) *States*: Compute the automaton abstract states as boolean conjunctions of the weakest pre-conditions (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'.

(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.

D SLS QUEUE SOURCE CODE

Below is the implementation of the Scherer *et al.* [Scherer III et al. 2006] queue. Path labels such as 1t or 1f are included to indicate which paths from Sec. 6.2 correspond to those program locations, where possible. (We have slightly refactored the second portion of the implementation in our path graph.)

```
1 public void enq(T e) {
2   Node offer = new Node(e, NodeType.ITEM);
3   while (true) {
4     Node t = tail.get(), h = head.get();
5     if (h == t || t.type == NodeType.ITEM) {
6       Node n = t.next.get();
7       if (t == tail.get()) {
8         if (n != null) {
9           tail.compareAndSet(t, n); 1t,1f
10        } else if (t.next.compareAndSet(n, offer) ) {
11            3t
12            tail.compareAndSet(t, offer); 3't,3'f
13            while (offer.item.get() == e); 3"
14            h = head.get();
15            if (offer == h.next.get()) {
16               head.compareAndSet(h, offer); return; 3"at,3"af
17            } else { return; 3"b }
18        } else { restart 3f }
19      } else { restart ② }
20    } else {
21      Node n = h.next.get();
22      if (t != tail.get() || h != head.get() || n == null ④) {
23         continue;
24      }
25      boolean success = n.item.compareAndSet(null, e); 5t,5f
```

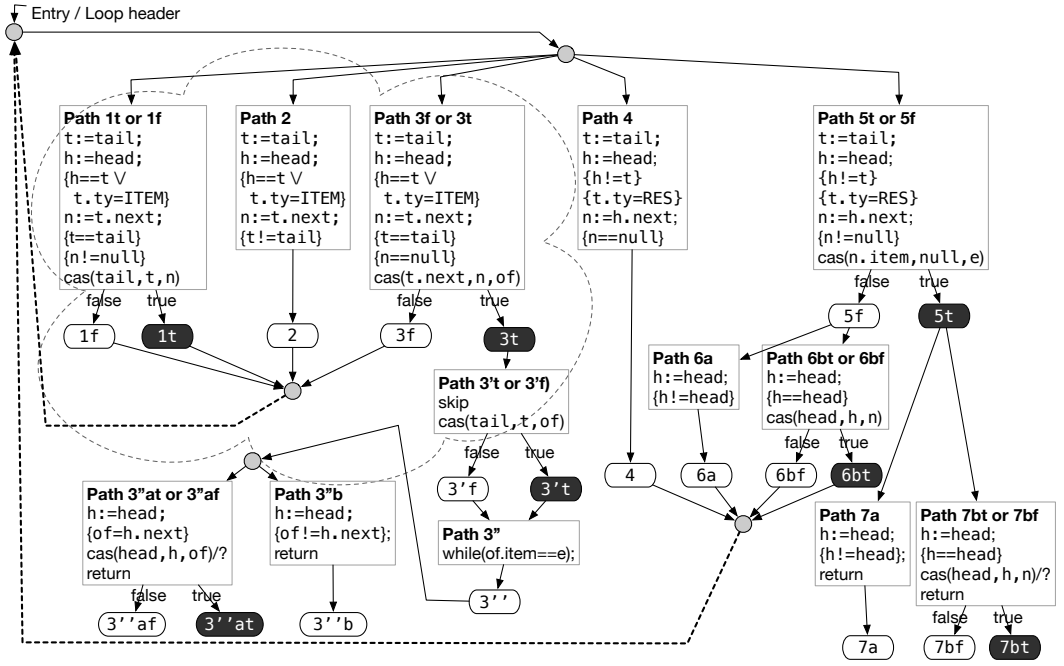


Fig. 7. (Reproduction of Fig. 11: The implementation of a synchronous queue due to Scherer III et al. [2006].)

```
26        head.compareAndSet(h, n); {6,7}bt,{6,7}bf
27        if (success)
28          return; 7a,7bt,7bf
29        else
30          restart 6a,6bt,7bf
31        }
32    }
33 }
```

D.1 SLS queue implementation graph

We describe the implementation in Fig. 11, beginning with the cloud-surrounded area in upper left-hand half of the diagram which is, essentially, the Michael-Scott queue. In this region the queue is a list of *items* (with a dummy head node), whereas the new portions of the implementation apply when the queue is a list of reservations. Paths 1t and 1f attempt to advance the tail pointer. Path 2 is interrupted by a recently changed tail pointer. Paths 3t and 3f attempt to swap tail's next to their new item offer node. If successful, paths 3't and 3'f attempt to advance the tail pointer.

Path 3" is the synchronous part of the algorithm: waiting for an enqueued item to be consumed by a dequeuer. At that point, the head pointer may be stale, and paths 3"at, 3"af and 3"b try to advance the head pointer.

Alternatively the queue may be a list of reservations, again with a dummy head node. Paths 5t and 5f attempt to fulfill a dequeuer's reservation by swapping null for an element. Path 5f is doomed to restart, while path 5t will soon return. In either case, the enqueue first attempts to advance the head pointer (paths 6a, 6bf, 6bt, 7a, 7bf, 7bt).

The implementation of dequeue is a sort of dual operation. When the queue is a non-empty list of *items*, dequeue tries to take the first item by swapping the head's next value for null (and then

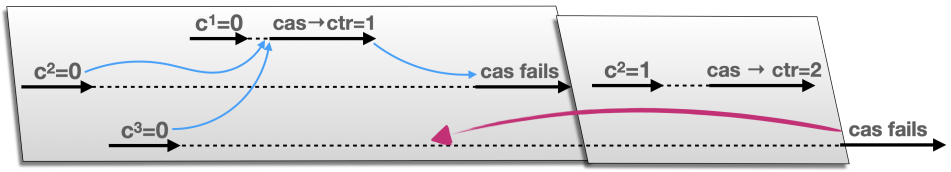


Fig. 8. An increment-only execution for which there is an equivalent representative execution (as suggested by the large wavy arrow) that is in the layer quotient.

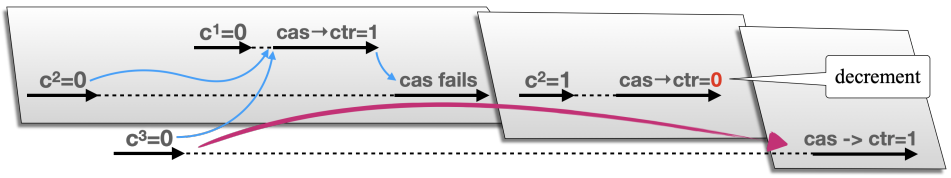


Fig. 9. An execution where the second thread executes a decrement, which is equivalent to the representative execution suggested by the wavy arrow.

tries to advance the head pointer). When the queue is empty or a list of reservations, dequeue redirects the tail's next to its new reservation node (and then tries to advance the tail pointer). After appending the reservation, dequeue spins until a value is swapped in, and then tries to advance the head pointer before returning.

E DETAILED EXPLANATION FOR COUNTER

To explain the equivalence between arbitrary interleavings of increment invocations and representative executions in quotient $\langle\lfloor O \rfloor\rangle$, we consider the execution pictured in Figure 8. This execution is *not* in $\langle\lfloor O \rfloor\rangle$ because the unsuccessful iteration of thread 3 is interleaved with *two* successful CASs: it reads `ctr` before the first successful CAS (in thread 1) and after the second successful CAS (in thread 2). Yet, as explained above, a layer interleaves an unsuccessful iteration with a *single* successful CAS.

However, the second read of `ctr`, corresponding to the unsuccessful CAS in thread 3, is enabled even if executed earlier just after the first successful CAS. Moreover, since retry-loop iterations are "forgetful", *i.e.,* there is no flow of data from one iteration to the next (the value of `ctr` is read anew in the next iteration), executing the unsuccessful CAS earlier would not affect the future behavior of this thread (and any other thread because it is a read) even if this reordering makes this unsuccessful CAS read a different value of `ctr` (value 1 instead of 2). This reasoning extends even when the iteration of thread 3 is interleaved with more than two other iterations.

The case of increment-only executions is simpler because it does not include the so-called ABA scenarios in which `ctr` is changed to a new value and later restored to a previous value. Every successful CAS will write a new value to `ctr` and will make all the other invocations that read `ctr` just before to restart.

Interleavings of increment and decrement invocations can exhibit the ABA scenario described above, as exemplified in Figure 9. The value 0 read by thread 3 before the first successful CAS is restored by the second successful CAS (performed in a decrement invocation). This execution is *not* a representative execution in $\langle\lfloor O \rfloor\rangle$ because the successful retry-loop iteration in thread 3 interleaves with other two successful iterations while in $\langle\lfloor O \rfloor\rangle$ executions, every successful iteration is executed in isolation w.r.t. other successful iterations. However, the equality test in the successful CAS of thread 3 (`ctr == 0`) is enough to conclude that the previous read can be commuted to the right and just before the CAS. This allows to group together the actions of the third layer and obtain a representative execution from $\langle\lfloor O \rfloor\rangle$, extended to include decrements. To this end, we introduce decrement layers of the form $[(\texttt{c:=ctr}^{n}_{inc} \cdot \texttt{c:=ctr}^{m}_{dec}) \cdot \texttt{c:=ctr;cas(ctr,c,c-1)/true} \cdot$

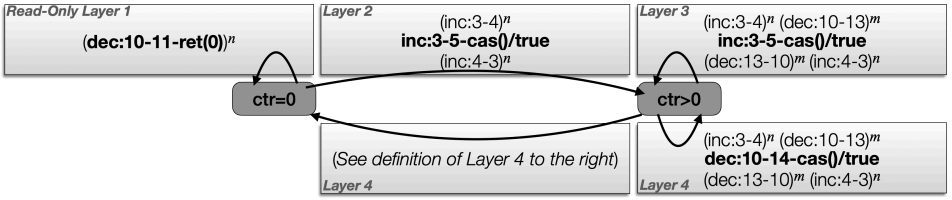


Fig. 10. An automaton representation of layer-serialized executions of the counter.

$(\texttt{cas/false}^m_{dec} \cdot \texttt{cas/false}^n_{inc})$. In this expression, n concurrent increment threads and m concurrent decrement threads interleave with a single successful decrement thread (we also subscripted with inc/dec to indicate where the action came from.). All unsuccessful threads' operations commute with each other and are put in a canonical form (later the interpretation of a^n will order a's according to thread ids). We similarly augment the increment layers with concurrent decrement threads.

Decrement invocations can also be formed exclusively of *read-only* iterations when they observe that `ctr` is 0. The last iteration in such invocations returns 0 and performs no write to the shared memory. Such loop iterations that read the value of `ctr` at the same time (after the same number of successful CASs) are grouped in a layer as well. They can be assumed to execute in isolation because they execute a single memory access.

F LAYER AUTOMATON FOR COUNTER

Overall, a quotient of the counter contains sequences of layers as described above. The order in which layers can occur in an execution can be constrained using regular expressions or equivalently, automata representations as shown in Fig. 10. In this *layer automaton*, states are properties of the shared memory that identify preconditions enabling shared-memory writes (successful CASs), and transitions represent layers.

This automaton consists of two states depicted in dark gray, distinguishing shared-memory configurations where the precondition of a successful CAS in decrement invocations (`ctr` > 0) holds. The self-loop on the initial state represents a layer (Layer 1) formed of an arbitrary number of decrement iterations returning value 0, executed by possibly different threads. "dec:10-11-ret(0)" refers to the control-flow path of decrement from Line 10 to Line 11 to return. This is just an abbreviation; formally it is represented with KAT expressions. Layer 2 occurs on the outgoing transition from the initial state and this layer is formed from a successful increment iteration interleaved with an arbitrary number of unsuccessful increment iterations executed by different threads (when `ctr` equals 0 all decrement retry-loop iterations reach the `return` statement). Iterations are represented as control-flow paths in the code of the methods. inc:3-5-cas()/true summarizes the single successful write path in the layer: an increment control-flow path that begins on Line 3, proceeds to the CAS, succeeds the CAS and returns. The final expression in Layer 2 summarizes an arbitrary number of threads failing the test on Line 4 (due to the successful write path), and loop back to Line 3. The outgoing transitions from the second state represent layers containing a successful increment (Layer 3) or decrement iteration (Layer 4), each interleaved with an arbitrary number of unsuccessful increment or decrement iterations. Finally, the transition from `ctr>0` to `ctr=0` involves the same Layer 4, despite landing in a new automaton state.

G QUOTIENT FOR MICHAEL-SCOTT QUEUE (FURTHER DETAILS)

The write operations in the layers induce the state changes as shown by the various edges in Fig. 1. For example, the *Dequeue Succeed Layer* can move from automaton state q_2 to automaton state q_1.

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). Other 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).

The layer quotient as represented in this layer-automaton is in some sense not optimal because some pairs of write paths commute, *e.g.,* enq writing to `Q.tail` and deq writing to `Q.head` when the queue is non-empty. However, in these circumstances the overall enq/deq commute in the sequential semantics of the object. Commuting these write linearization points in the layers corresponds to commuting the overall methods. Consequently, the layer quotient can be seen as optimal modulo method-level commutativity.

Theorem G.1. *The above layer automaton is an abstraction of a quotient for Michael-Scott Queue.*

Proof by the methodology of Def. 5.6. (WPC) For the deq successful CAS on `Q.head` and adv successful CAS on `Q.tail`, old reads are not possible because every CAS changes those pointers to fresh values. Thus, if the CAS was successful, the read must be in the current layer (there are no other successful CASs in between). The enq CAS on `Q.tail->next` is similar to Treiber's stack: `Q.tail->next` is only written once, so any old value of `Q.tail->next` will have the same value in the current layer so an old read can move to the current layer. Furthermore, if there was an old read of `Q.tail`, the value of `Q.tail` could not have changed without `Q.tail->next` first having been changed.

(LPC) 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.

Note that the Advancer Succeed Layer in reality cannot self-loop from q_3 because an invariant of the MSQ is that `Q.tail` can only lag behind by one link in the list. This happens because weakest preconditions from a *true* postcondition are over-approximate and did not include complicated invariant reasoning to accurately express the single-link lag condition. Consequently this layer automaton (soundly) over-approximates the executions of the MSQ.

H QUOTIENT FOR THE SLS QUEUE

Implementation. The implementation of the SLS queue is illustrated in Fig. 11 (the source code is given in Apx. D). This diagram is like a control-flow graph (entry point, loop header, branch/merge points, etc.), but with some flattening to make paths more explicit. Paths are identified where they end, with write paths denoted as **1t** and local paths as ④. Two paths that share a prefix and differ only based on a CAS result are denoted with a single box, but with true/false exit arcs, *e.g.,* **1t** and ①f. Later below we will write **D1t** versus **E1t** when we are referring specifically to dequeue versus enqueue. This is the source for enqueue (which appends item nodes or fulfills reservation nodes) and the source for dequeue is identical (except dequeue appends reservation nodes or consumes item nodes).

SLS, like MSQ, involves manipulating a list of nodes that are *items* with a dummy head node. There is a synchronous blocking on path ③". However, for SLS, alternatively the queue may be a list of reservations, and the right-hand paths attempt to fulfill a dequeuer's reservation by swapping null for an element. The implementation of dequeue is a sort of dual, omitted for lack of space. Below we denote paths such as **D5t** to mean the dequeue dual of enqueue's **5t**. Note that, unlike

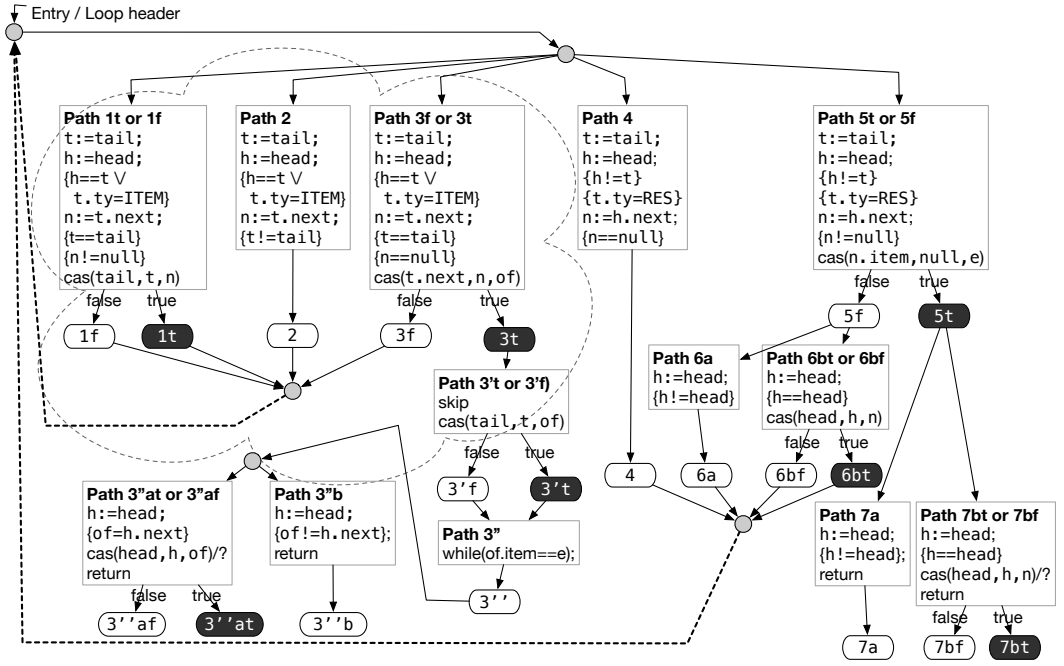


Fig. 11. The implementation of a synchronous queue due to Scherer III et al. [2006].

Treiber's stack or the MSQ, in the SLS queue a method invocation could involve a series of paths, *e.g.*, the sequence 3t; 3't; 3"; 3"at, that involves multiple write operations.

The cloud-surrounded area in the upper left-hand half of the diagram is essentially MSQ and it involves manipulating a list of nodes that are *items* with a dummy head node. There is a synchronous blocking on path 3". Alternatively the queue may be a list of reservations, and the right-hand paths attempt to fulfill a dequeuer's reservation by swapping null for an element.

The SLS queue demonstrates that a method implementation could consist of sequentially composed paths which define different layers. As we will see, advancing the tail pointer (and the head pointer) are subpaths of method implementation. Moreover, the synchronous behavior involves busy-wait/blocking during the implementation, after which point further paths are executed.

Quotient. The quotient for SLS is discussed in Sec. 6.2. In this appendix, we show Fig. 12 which is similar to the Sec. 6.2 quotient, but with precise CFG locations in the layer definitions.

Technically the states require one further predicate to indicate whether there is currently a thread at location 3t, omitted for lack of space. This is needed because the subsequent paths use the local variable `t` which is an old read done in the previous path. This is acceptable because it is only possible that one thread can be at location 3t so the old read is still valid during the subsequent path. Typically we have found that implementations do not perform such "old reads" which are only correct as a result of very delicate reasoning.

Theorem H.1. *The SLS queue is linearizable.*

Proof: The following expression uses the same layers, some marked E or D for linearization points:

$$\begin{array}{lll} (& [\mathsf{Dapp} \cdot \mathsf{TA} \cdot (\mathsf{Eswap}_{E;\,D} \cdot \mathsf{HR} \;\cdot\; \mathsf{Dapp} \cdot \mathsf{TA})^* \cdot \mathsf{Eswap}_{E;\,D} \cdot \mathsf{HR}] & \text{// LHS} \\ + & [\mathsf{Eapp}_E \cdot \mathsf{TA} \cdot (\mathsf{Dswap}_D \cdot \mathsf{HR} \;\cdot\; \mathsf{Eapp}_E \cdot \mathsf{TA})^* \cdot \mathsf{Dswap}_D \cdot \mathsf{HR}] & \text{// RHS} \\ )^* & \cdot \;\; (\; [\mathsf{Dapp} \cdot \mathsf{TA}]^* \;\; + \;\; [\mathsf{Eapp}_E \cdot \mathsf{TA}]^*) & \end{array}$$

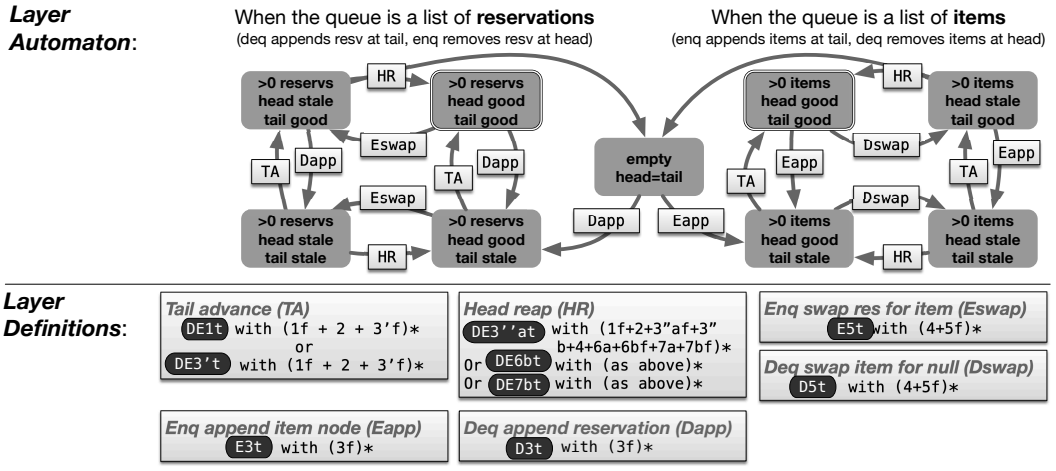


Fig. 12. Layer automaton for the synchronous SLS queue. Layers' acronyms and their definitions are given in the lower half of the figure. For conciseness, layer definitions do not split the prefix/suffix of the read paths.

This expression captures iterating through the left and righthand sides of the automaton (passing through the empty ADT state in between), followed by either unmatched appended dequeue reservations or unmatched appended enqueue items. When the queue consists of reservations, the Eswap layer provides the linearization point for enqueue, but also the corresponding dequeue. TA and HR layers are positioned next to a corresponding app and swap (resp.).

We thus prove (#1) This expression is an abstraction of the quotient: by induction on any execution, feasible actions can be reordered into layers and those layers can be ordered into the above expression. (#2) For linearizability, we project out the LP operations to obtain simply $(E{\cdot}D)^* \cdot (E^*{+}D^*)$. Thus, combining with #1, all executions meet the sequential spec. of a queue.

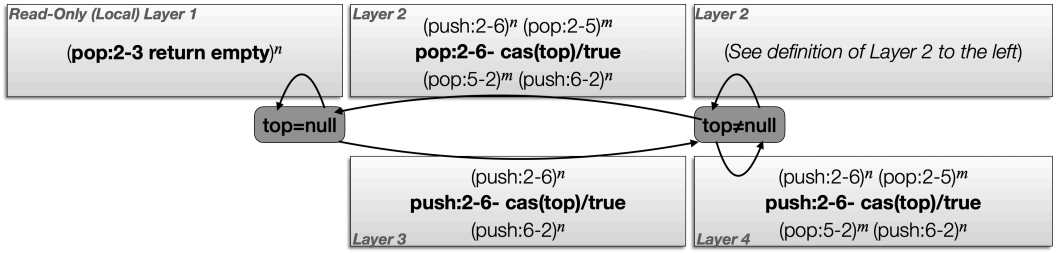
I QUOTIENT FOR TREIBER'S STACK

Recall the implementation of Treiber's stack [Treiber 1986], stored as a linked list from a global pointer top, and manipulated as follows:

```
1 void push(int item){ while(1){
2   node_t* n = malloc(...);
3   n->val = item;
4   node_t* oldTop = top;
5   n->next = oldTop;
6   if(CAS(top,oldTop,n) ret;
7 } }
```

```
1 int pop(){ while(1){
2   node_t* oldTop = top;
3   if(oldTop==NULL) { ret 0; }
4   newTop = oldTop->next;
5   if(CAS(top,oldTop,newTop) ret oldTop->val;
6 } }
```

The states for the layer-automaton of the Treiber's Stack (derived from the pre-conditions of successful push and pop operations) are simply top=null and top≠null. The Treiber stack can thus be decomposed into a layer automaton as follows:



Above the automaton states are given in rounded dark boxes, and edges are labeled with layers. We abbreviate local paths using source code line numbers rather than KAT expressions. For example `pop:2-5` means the path starting at the beginning of Line 2 of pop and proceeding to the beginning of Line 5. Layer 1 is a local layer, in which the state is `top=null`. In this layer, there is only one local path from pop that is enabled for some n threads and it pertains to returning 0 to indicate empty. Layer 2 occurs from a state where `top≠null` and the pop ARW action for the compare-and-swap occurs, causing n other pushes' and m pops' CAS attempts to fail (on their lines 6 and 5, respectively) and thus they restart (transition back to their respective Line 2s). The write path is in bold. The other layers are similar, with a single pop or push ARW invalidating other pop/push attempts. Layer 2 occurs as a label in two different transitions. Layer 5 self-loops at state `top≠null`, which abstracts over all non-empty stacks.

Lemma I.1. *The above layer automaton is an abstraction of a quotient for Treiber's stack.*

Proof: By the methodology of Def. 5.6. Per WPC, we must show that an old read of `top` and `top->next`, with then arbitrarily many write paths interleaved, can be moved to the right just before the successful CAS (an unsuccessful CAS belongs to a local path, discussed next). The successful CAS checks that `top` is unchanged since the old read. Moreover, since `top->next` is only written once, if `top` is unchanged then `top->next` must also be unchanged[6]. Therefore both old reads could be moved to the right just before the successful CAS, and a whole write path can be assumed to execute without interruption.

Per LPC, requiring that each local path, pops returning 0 or iterations with failed CASs, can be re-ordered to interleave with at most one write path. Iterations where a pop returns 0 perform a single access to shared-memory (reading `top`) and therefore, they can be assumed to execute without interruption. The failed CAS in an iteration can be re-ordered to occur just after the first successful CAS that follows the read of `top` in the same iteration. This holds because in Treiber's stack successful CAS operations always mutate `top` to a fresh value (assuming memory freshness).

J QUOTIENT FOR ELIMINATION STACK

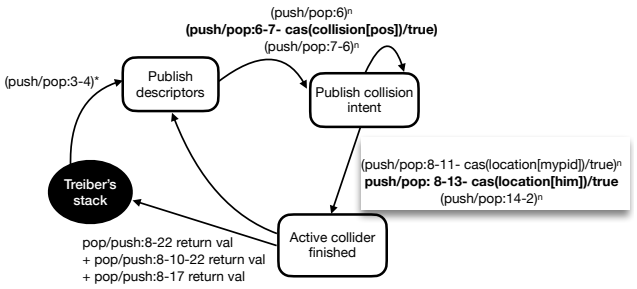
The Elimination Stack [?] augments Treiber's stack with a protocol for "colliding" push and pop invocations so that the push passes its input directly to the pop without affecting the underlying data structure. An invocation starts this protocol after performing a loop iteration in Treiber's stack and 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 object 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.

[6]We assume a semantics modeling garbage collection where memory cannot be reallocated. Without this assumption, it is possible that `top` is unchanged, but `top->next` has changed. This is known as an ABA bug.

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 4–7). 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 11) and modify the cell of the other thread (line 12) 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 18–20). Failing the second CAS leads to a restart (line 15). If the invocation reads a NULL thread id from `collision`, then it tries to clear its cell before restarting (line 22). If it fails, then as in the previous case, a collision happened with a third thread and the current thread can simply return (line 23–25).

```
1 void push/pop(descriptor p){ while(1) {
2     one iteration of Treiber stack
3     location[mytid] = p;
4     pos = nondet();
5     do {
6       him = collision[pos]
7     } while (!CAS(&collision[pos], him, mytid))
8     if him != NULL {
9        q = location[him]
10       if ( q != NULL & q.id = him & p.op != q.op ) {
11          if (CAS(&location[mytid],p,NULL)) {
12             if ( CAS (&location[him], q, p/NULL) )
13                return NULL/q.input
14             else
15                continue
16          }
17          else {
18             val = NULL/location[mytid].input;
19             location[mytid] = NULL;
20             return val
21    }  }  }
22    if (!CAS(&location[mytid],p,NULL)) {
23       val = NULL/location[mytid].data;
24       location[mytid] = NULL;
25       return val
26 } } }
```

We use the automaton below to describe a sound abstraction of the quotient. Layers of Treiber's stack (defined in Section I) interleave with layers of the collision protocol (some components are not exactly layers as in Definition 5.2, but very 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 top right 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 8 or 10. 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 such serialized collisions).

Theorem J.1. *The above automaton is an abstraction of a quotient for the Elimination Stack.*

Proof: (Sketch) We need to show that every execution of the Elimination stack is equivalent to some execution represented by this automaton up to reordering of commutative actions. The interactions in the Treiber's stack component do not interfere with collisions (they use disjoint addresses in the shared memory) and therefore every execution can be assumed (up to commutativity) to execute in phases as follows: some number of invocations executing a sequence of layers as in the Treiber's stack layer automaton (competing on the `top` pointer) followed by some number of invocations trying to collide with each other, followed again by Treiber's stack layers, and so on. In the following we show that the collisions can be reordered to occur serially as in the above automaton.

We proceed by induction on the number of successful CASs at line 12 (the second CAS required for a successful collision). Consider the first such successful CAS, denoted as CAS_2/T and let $\mathcal{F}_2$ be the set of threads whose next step in the execution after this point is a failed CAS on the same address. As in previous proofs, all these failed CASs can be reordered (to the left) to occur immediately after the successful one. Then, by the control-flow of an invocation, all threads in $\mathcal{F}_2$ executed the successful CAS at line 11 before their failed CAS. All these successful CASs turn the `location` cell of those threads to NULL. Since no other thread (besides themselves) can turn it back to some non-NULL value (see the test at line 10), they can be reordered to occur immediately before CAS_2/T. This leads to an interleaving around CAS_2/T that conforms to the expression that labels the transition leading to "Active collider finished". Then, looking at other steps before CAS_2/T, for every successful CAS on a `collision` cell, one can construct a layer as the one labeling the transitions leading to "Publish collision intent" and also serialize the steps 3–4 for every thread. This is possible because all these interactions concern different memory addresses. Finally, CAS_2/T wrote on the `location` cell of a thread `him`, and no other thread can modify this value until `him` reads it, observes to have been collided and returns (CAS_2/T writes either NULL to `location[him]` in which case the first conjunct at line 10 will fail in another thread, or a descriptor with an `id` field different from `him` in which case the second conjunct at line 10 will fail). Therefore, all those steps of `him` can be reordered to the left to occur immediately after the interaction around CAS_2/T, which completes the handling of this first collision. The subsequent collisions can be handled in a similar manner.

K QUOTIENT FOR RDCSS

The Restricted Double-Compare Single-Swap (RDCSS) [**?**] is a restricted version of a double-word CAS (acting atomically on two addresses) which modifies a so-called data location provided that this location and another so-called control location have some given expected values. This is an instance of an atomic read-modify operation, i.e., the tests and the write should happen atomically. It is assumed that data and control locations are disjoint (i.e., the same address can not be a data address in some invocation and a control address in another). The code of the main RDCSS operation is given below (for simplicity, we omit the read operation):

```
1 void RDCSS(descriptor *d){
2   do {                                          9  void Complete(descriptor *d) {
3      r = CAS(d->DATA_ADDR, d->exp_data, d);    10    if ( * d->CONTROL_ADDR == d->exp_control )
4      if ( isDescriptor(r) ) Complete(r);       11       CAS(d->DATA_ADDR, d, d->new_data);
5   } while ( isDescriptor(r) )                  12    else
6   if ( r == d->exp_data ) Complete(d)          13       CAS(d->DATA_ADDR, d, d->exp_data)
7   return r;                                    14  }
8 }
```

The inputs of the operation are put inside a `descriptor` structure: `DATA_ADDR` and `CONTROL_ADDR` are the data and control addresses, respectively, `exp_data` and `exp_control` are the expected values of these addresses, and `new_data` is the new value to be written to the data address (provided that the data and control addresses store the expected values).

`RDCSS` attempts a standard CAS on the data address to change the old value into a pointer to the descriptor (line 3). This CAS checks that the data address has the expected value, and if it fails, the operation simply returns. In the context of this implementation, we assume that a CAS returns the value of the location before any modification (if any) and not just a Boolean. If the CAS succeeds, then the operation calls `Complete` in order to check the control location and finalize the modification if possible (line 6). `Complete` checks the value of the control location and if it has the expected value, then it attempts a CAS to change the data address (line 11); note that the data address currently stores a pointer to a descriptor. Otherwise, it attempts a CAS to revert the data address to its old value (line 13).

When multiple threads compete to change the same data address, it may happen that the thread succeeding the first CAS at line 3 (the initiator) is slow and before it executes the call to `Complete`, another thread fails its CAS but finds a descriptor at this address (it is assumed that descriptor pointers can be distinguished from data values). Then, this other thread will try to help the slower one and call `Complete` itself (line 4). Note that all the information needed to help the slower thread is stored in the descriptor.

We use the expression below to describe a sound abstraction of the quotient:

// successful modification

$\Big($ **3-CAS/true** $\cdot$ (3-CAS/false-4)^n $\cdot$ { **4-6-11-CAS/true** $\cdot$ (4-11-CAS/false)^n

+ **3-CAS/false-4-11-CAS/true**$\cdot$ (4-11-CAS/false)^n $\cdot$ 4-6-11-CAS/false }

+

// fail: wrong control value

3-CAS/true $\cdot$ (3-CAS/false-4)^n $\cdot$ { **4-6-13-CAS/true** $\cdot$ (4-13-CAS/false)^n

+ **3-CAS/false-4-13-CAS/true**$\cdot$ (4-13-CAS/false)^n $\cdot$ 4-6-13-CAS/false }

+

// fail: wrong data value

(3-CAS/false)* $\Big)^*$

Executions in the quotient are iterations (note the outer *) of three types of "phases" (note the outer union and read expressions from top to bottom): (1) a phase in which the data address is modified successfully (without or with help), (2) a phase in which the modification fails because the *control* address does not have the expected value (noticed by the initiator of the modification or a helper thread), and (3) a phase in which the modification fails because the *data* address does not have the expected value.

The first two phases have a common prefix: some initiator thread succeeding the CAS at line 3 and some number n of threads failing the same CAS and reading the descriptor written by the initiator. Next, for the first phase, there are two cases: (1) the initiator succeeds the second CAS at line 11 (after calling `Complete` at line 6), and those n threads will fail the same CAS (after calling `Complete` at line 4), or (2) some helping thread which fails the same CAS as the other n threads

will succeed the CAS at line 11 (after calling `Complete` at line 4), and the initiator together with the other n threads fail the same CAS. For the second phase, there are two analogous cases in which either the initiator or a helping thread observes a wrong value for the control location and succeeds the CAS at line 13 to revert the value of the data location. The third phase is trivial and consists of an arbitrary number of failed instances of the CAS at line 3.

THEOREM K.1. *The above expression is an abstraction of a quotient for RDCSS.*

Proof: (Sketch) Since steps of `RDCSS` invocations on different data addresses commute (the assumption that data and control addresses are disjoint is important here), we focus on invocations that act on the same data address. We follow the same strategy as for Elimination Stack, and proceed by induction on the successful CASs in `Complete` (line 11 or line 13). Consider the first such CAS and assume that it is at line 11. This corresponds to the first phase above and the case of line 13 which corresponds to the second phase can be handled similarly. There are two cases to consider:

- The thread t performing this CAS called `Complete` at line 6. If there are threads whose next step in the execution after this point is a failed CAS on the same address and expecting to find the same descriptor, then all of these steps can be reordered to the left to occur immediately after the successful one. By control-flow, these other threads arrived there by calling `Complete` at line 4 which means that they fail their CAS at line 3 and they read the same descriptor. All of these failed CASs can be reordered to occur immediately after t succeeding its CAS at line 3. Overall, these reorderings lead to an execution fragment with the shape described in the first line of the expression above.
- The thread t performing this CAS called `Complete` at line 4. Following a similar reasoning while taking into account that another thread t' initiated this modification by succeeding a CAS at line 3, one can reorder steps to obtain a prefix with the shape given by the second line of the expression above.

While building serializations of phases of type (1) and (2) above, any failed CASs at line 3 that return the same value can be reordered to occur one after another, thereby creating phases of type (3). And these phases of type (3) can occur "outside" of phases of type (1) and (2) since they have no effect on the shared memory.

L QUOTIENT FOR THE LIST SET

We here consider a List Set Object and describe the layer expressions and proof that they are an abstraction of the List Set's quotient. This example is a Set object implemented as a sorted linked list [O'Hearn et al. 2010], which involves a read-only traversal `locate`, and then small atomic sections to link/unlink nodes (for `insert`/`delete`, respectively). `locate` traverses the list from the `head` and returns a pair of nodes (`x`,`y`) such that `y` has the key of interest or else `x` points to the last node whose key is below `k`. It is implemented as a loop that may perform an unbounded number of shared-memory reads. We assume that it is abstracted with the postcondition at line 3 in `insert` stating that `y` is the successor of `x`, the input `k` is in between `x.key` and `y.key`, and that at some point between the invocation of the operation and "now", `x` resides on a valid search path for `k` that starts at the head of the list, denoted as $\diamond\texttt{head} \xrightarrow{\texttt{k}} \texttt{x}$. Recent work [Feldman et al. 2018, 2020] shows that this postcondition can be derived easily by showing that roughly, list nodes are never updated once they become unreachable. Therefore, the implementations of `insert` and `delete` are as follows:

```
1 int insert(int k) { while(1) {
2     struct node_t *z = ...;
3     assume x.next = y ∧ x.key < k ≤ y.key ∧ ◇head -k-> x
4     atomic {
5       if (x->next == y && x->del == 0) {
6         if (y->key != k) {
7           z->next = y;
8           x->next = z;
9           return 1;
10        } else { return 0; }
11    }
12 } }
```

```
1 int delete(int k) { while(1) {
2   assume x.next = y ∧ x.key < k ≤ y.key ∧ ◇head -k-> x
3   atomic {
4     if(x->next == y && x->del == 0) {
5       if (y->key == k) {
6         y->del = 1;
7         x->next = y->next;
8         return 1;
9       } else { return 0; }
10    }
11  }
12 } }
```

The `insert` method will link a node `z` in between `x` and `y`, provided that `k` wasn't already in the list. The `delete` method returns 0 if the element wasn't in the list and otherwise, marks node `y` for deletion, and then updates `x` to skip past node `y`. The `delete` method marks deleted nodes with a `del` flag before they are unlinked. Because `delete` marks deleted nodes' `del` fields, a concurrent `locate` that has just found this node, but was then preempted by `delete`, will return a node that's marked as deleted and unlinked, not simply unlinked.

As we discuss below, for List Set the layer expressions based on interleavings of two threads generalizes to arbitrary threads. We thus define the states of the automaton in terms of the possible values from the perspective of one reader and one writer. In these states below, x_w denotes the writer's `x`, x_r denotes the reader's `x` and similar for the other variables. The `x` and `y` variables are existentially-quantified in the pre-conditions because they are method-local variables and not inputs. We omit the sub-formula $\diamond\text{head} \xrightarrow{k} x$ because this condition does not affect the enabled status of a layer.

$$
\begin{aligned}
q^1 &= [\![\exists x_r,y_r.\ x_r\text{->next}=y_r \wedge \text{!}x_r\text{->del} \wedge k_r=y_r\text{->key}]\!],\\
q^2 &= [\![\exists x_r,y_r.\ x_r\text{->next}=y_r \wedge \text{!}x_r\text{->del} \wedge x_r\text{->key} < k_r < y_r\text{->key}]\!],\\
q^3 &= [\![\exists x_r,y_r,x_w,y_w.\ x_w\text{->next}=y_w \wedge \text{!}x_w\text{->del} \wedge x_w\text{->key} < k_w < y_w\text{->key} \wedge\\
&\quad x_r=x_w \wedge x_r\text{->key} < k_r < y_r\text{->key}]\!],\\
q^4 &= [\![\exists x_r,y_r,x_w,y_w.\ x_w\text{->next}=y_w \wedge \text{!}x_w\text{->del} \wedge x_w\text{->key} < k_w < y_w\text{->key} \wedge\\
&\quad x_r=x_w \wedge k_r=y_r\text{->key}]\!],\\
q^5 &= [\![\exists x_r,y_r,x_w,y_w.\ x_w\text{->next}=y_w \wedge \text{!}x_w\text{->del} \wedge k_w=y_w\text{->key} \wedge x_r=x_w \wedge\\
&\quad x_r\text{->key} < k_r < y_r\text{->key}]\!],\\
q^6 &= [\![\exists x_r,y_r,x_w,y_w.\ x_w\text{->next}=y_w \wedge \text{!}x_w\text{->del} \wedge k_w=y_w\text{->key} \wedge x_r=x_w \wedge k_r=\text{key}]\!],\\
q^7 &= [\![\exists x_r,y_r.\ x_r\text{->next}=y_r \wedge x_r\text{->del} \wedge x_r\text{->key} < k_r \leq y_r\text{->key}]\!]
\end{aligned}
$$

With these 7 states, 2 write paths (one from `insert`, one from `delete`) and 6 read paths, there are many transitions to consider, although many of them are labeled with the same layer. In fact, the List Set can be decomposed into 8 layers, enumerated below. For lack of space, we omit the automaton, but the definitions, including all 77 feasible transitions, can be seen in the output of our

tool (which we discuss in the next section) shown in Apx. A. Below we refer to example transitions in Apx A, denoted δ_i.

(1) A layer with a `delete` write path that updates `x.next` to point to `y.next`, causing one `insert` and one `delete` path to fail when finding `x.next≠y`.(*e.g.,* δ_2)
(2) A layer with an `insert` write path that updates `x.next` to point to `z`, causing one `insert` and one `delete` paths to fail when finding `x.next≠y`. (*e.g.,* δ_9)
(3) A local layer consisting of one `delete` path, when the key is not in the set. (*e.g.,* δ_{31})
(4) A local layer consisting of one `insert` path, when the element is already in the set. (*e.g.,* δ_{47})
(5) Four local layers consisting of `insert` or `delete` paths when the node `x` is already marked for deletion. (*e.g.,* δ_{63})

Note that `insert` and `delete` have more than one control-flow path that "fails" because of the nested conditional inside the atomic read-write.

As in the Michael/Scott queue, here again the layer-quotient is optimal *modulo* method-level commutativity. At the method-level, operations such as insertion/deletion of different elements commute and their corresponding linearizations can be commuted (different orders of write layers) in the layer quotient.

Lemma L.1. *The above layer automaton is an abstraction of a quotient for the List Set.*

Proof by the methodology of Def. 5.6. To prove the lemma we first note that the post-condition of `locate` ensures that `x` was reachable and that `y=x->next`. In all read and write paths, the ARW checks that `y=x->next` still holds. Furthermore, an invariant of the implementation is that if `x` was reachable at some point in the past (*i.e.,* when `locate` executed) and `!x->del` holds in the atomic section, then `x` is still reachable in the atomic section (this holds because elements are marked before being unlinked). Therefore, if `locate`'s postcondition was true in the past, it remains true when the ARW succeeds and the `assume` can be reordered to occur just before it. For local paths, as in previous cases, a failed ARW can be commuted to the left to occur just after the first ARW that modifies the location `x`.

M QUOTIENT FOR THE HERLIHY-WING QUEUE

Recall the queue due to Herlihy and Wing [Herlihy and Wing 1990], reproduced below:

```
1 int deq() { while(1) {
2   assume 0 <= range < back; int j = 0;
3   while(j<range) {
4     v := swap(items[j],null);
5     if (v != null) return v;
6     j++; }
7 } }
```

```
1 void enq(int v) {
2    i := back++;
3    items[i] = v;
4 }
```

Enqueue (on the right) reserves the next slot in the array `items` by *atomically* reading and incrementing the shared variable `back`, and then assigns the value to that slot in a *second* write to the shared state. Meanwhile, dequeue (on the left), in an outer loop reads into `range` any value strictly smaller than `back` and then iterates from 0 to `range`, looking for a slot containing an item to atomically dequeue. For every `j`, it atomically reads `items[j]` into `v` and writes `null` (written as a `swap` instruction), and if the read value is not `null`, it returns it. This is actually a sound abstraction of the original version which assigns `back-1` to `range` instead of any smaller value. Soundness follows easily from the fact that reading a smaller value will only make the dequeue restart more often (perform more traversals in which there is no occupied slot), but not affect safety. In the reasoning below, we will use the fact that such a non-deterministic read commutes to the right of any increment of `back` (it is a right mover).

We show that the Herlihy-Wing queue quotient can be abstracted by an expression given below. To this end we first, as below in Apx. M.1, prove that any iteration of the dequeue's outer `while(1)` loop can be considered atomic, modulo commutative re-orderings. Consequently, there exists a quotient of Herlihy and Wing Queue where outer loop iterations in dequeue are atomic sections. Since `items[i] = v` steps in enqueues commute (they write on different slots of the array), there exists a quotient where additionally, every sequence of `items[i] = v` steps before a dequeue iteration that is successful (contains a non `null` swap) is ordered w.r.t. the array slots that they write. The following expression is an abstraction of such a quotient: $(\texttt{deqF}^* \cdot (\texttt{enqI})^+ \cdot \texttt{enqW}^* \cdot \texttt{deqT}^*)^*$ where `enqI` and `enqW` denote the statements `i:=back++` and `items[i] = v` in enqueue, respectively, and `deqT` and `deqF` represent entire iterations of the outer loop in dequeue that end in a `return` and restarting the loop, respectively. The interpretation of $\texttt{enqW}^*$ is refined to be a set of sequences of labels `items[i] = v` (with thread ids) that are ordered w.r.t. the position i that they write to. Above, we also use straightforward feasibility arguments like "enqueues increment `back` before writing to `items`," and "a `deqT` must be preceded by a write to `items` in an enqueue."

Theorem M.1. *The above expression is an abstraction of the HWQ quotient.*

Theorem M.2. *The HWQ is linearizable.*

The set of traces represented by this expression admits a "simple" linearization point mapping which identifies `enqW` and `deqT` steps with linearization points of enqueues and dequeues, respectively. The restriction to traces in this quotient is instrumental for such a simple linearization point mapping. For arbitrary traces, the Herlihy and Wing Queue is known for having linearization points that depend on the future and that *can not* be associated to fixed statements, see e.g. [Schellhorn et al. 2012] !

M.1 Proof of atomicity of outer loop

We prove that any iteration of the outer `while(1)` loop can be considered to be an atomic section, modulo re-orderings of commutative actions. That is, there exists a quotient of this object formed of traces where all steps of such an iteration occur consecutively one after another.

We proceed by induction on the number of steps executing the swap at line 4 in dequeue and that find a non-`null` value in `items[j]`. In the base case, i.e., the number of such steps is 0, all the swaps at line 4 in all dequeue invocations find `null` values. Therefore, any possible write to an `items` slot (in enqueues) can be re-ordered after all swaps. Now all steps in the same outer loop iteration of a dequeue (the non-deterministic read of `back` and swaps returning `null`) can be re-ordered to occur consecutively one after another. In particular this relies on the fact that the non-deterministic read of `back` can return the same value even if executed after more increments of `back`. For a trace with $n + 1$ swaps returning non-`null` values, we focus on the first such step. Assume that it is a swap on some position `k`. All the writes to `items` slots strictly before `k` can be re-ordered to the right of this first non-`null` swap. This relies on the fact that all the other previous swaps return a `null` value and anyway, do not "observe" these writes. Similarly to the base case, all steps in the same outer loop iteration of a dequeue that completes before this first non-`null` swap (including) can be re-ordered to occur consecutively one after another. We are again using the fact that swaps read `null` values and there is no more write on the slots that they read. Now, removing the write to `items[k]` in enqueue and the dequeue iteration that removes this value from the current trace, we get another feasible trace that has n non-`null` swaps, for which one can apply the induction hypothesis.

N EVALUATION: ALGORITHM AUTHORS' CORRECTNESS ARGUMENTS

As discussed in Sec. 1, our goal is to provide a formal foundation for the scenario-based correctness arguments found in the literature. In this section, we evaluate our work by revisiting various such arguments in the literature, and comparing them with the quotient-based proofs presented in this paper. At the high level, our comparison shows that quotients make scenario-based reasoning more explicit and ensure that all cases are considered.

N.1 Treiber's Stack

Treiber's stack is fairly straight-forward. As such, it provided a good starting point for defining quotients yet the prose correctness arguments are fairly minimal. For example, the following is a comment on linearizability:

> *The linearization point of both the push() and the pop() methods is the successful compareAndSet(), or the throwing of the exception in case of a pop() on an empty stack.* – Herlihy and Shavit [2008b]

This prose identifies specific linearization points as (1) the "successful compareAndSet" and (2) the not-found exception. These LPs correspond to the layers in the quotient shown in Apx. I. Layers 2, 3, 4 are "successful compareAndSet" linearization points, and read-only Layer 1 is the linearization point for the not-found exception.

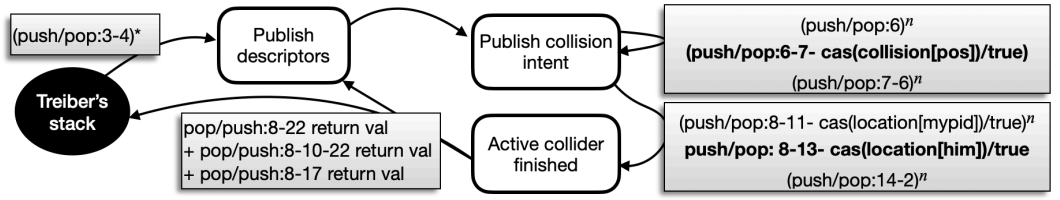
Summary. The following table summarizes the various elements of the correctness argument/proof, and identifies examples of where they occur in the Herlihy and Shavit [2008b] proof, and where they occur in the quotient proof.

Proof Element	Herlihy and Shavit [2008b] **Proof**	**Quotient Proof**
ADT states	"empty stack"	ADT states, e.g. (`top=null`)
Concurrent threads	(general description)	Superscripting $(\ldots)^n$
Thread-local step seq.	"try to swing [top] ... if [] succeeds, push() returns, and if not, the [] attempt is repeated"	Layer paths, e.g., `push:2-6`
Linearization pts.	"The linearization point of both the push() and the pop() methods is the successful compareAndSet(), ..."	The successful CAS in Layers 2, 3 and 4.
(continued)	"...or the throwing of the exception in case of a pop() on an empty stack."	Read-Only Layer 1

The layer quotient and, especially, the layer automaton (shown in Apx. I) helps make the Herlihy and Shavit [2008b] proof more explicit. The layer automaton makes the ADT states explicit. From each ADT state, one can consider which (i.e. all possible) layers are enabled, and which target states are reached via those layers. Linearization points are explicit in the layer quotient, occurring once with each layer transition. The layer quotient automaton also has the benefit of explicitly showing all of the linearizable executions: i.e. all the possible runs of the automaton. This is left as implicit in the Herlihy and Shavit [2008b] proof.

N.2 Elimination Stack

Section 5 of **?** gives a correctness proof for the elimination stack. We now review the proof and compare it with the quotient given in Apx. J. For reference, the following is a replication of the quotient automaton:



> *We note that a set is a relaxation of a stack that does not require LIFO ordering. We begin by proving that our algorithm implements a concurrent set, without considering a linearization order. We then prove that our stack implementation is linearizable to the sequential stack specification of Definition 5.1. Finally we prove that our implementation is lock-free.* – **?**

> *We now prove that our algorithm has correct set semantics, i.e. that pop operations can only pop items that were previously pushed, and that items pushed by push operations are not duplicated. This is formalized in the following definition [omitted Set semantics for methods Push/Pop]* – **?**

? decompose the proof into first Set semantics and then ordering considerations. In the quotient this is unnecessary because the layers capture the ordering and the elements in them. In the bottom right 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. (Note that the quotient automaton could also have been written in a more verbose way where the bottom right 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.) As discussed below, the thread that succeeds its CAS in the bottom right later is referred to as the "active" thread, and the thread with which the active thread collides is referred to as "passive." These concepts are explicit in the quotient: the thread taking the bold action in the bottom right is the "active" thread, and the thread that finds itself collided with in the layers on the arcs that exit the "Active Collider Finished" state, are "passive."

We now continue to examine the authors' proof:

> *In the following, we prove that operations that exchange their values through collisions are also correct set operations, thus we show that our algorithm has correct set semantics. ... We say that a colliding operation op is active if it executes a successful CAS in lines C2 or C7. We say that a colliding operation is passive if op fails in the CAS of line S10 or S19. [underlines added]* – **?**

The authors lay out a few definitions, which are also captured by the layer quotient. Above the authors' intuitive concept of "active" is captured by the paths in a layer that succeed their CAS. Likewise for "passive" and CAS failure. As mentioned above, the active thread is captured as the bold thread that succeeds its CAS in the bottom right layer; the passive thread is the thread that finds itself collided with in the layers on arcs exiting the bottom right layer.

> *We say that op is trying to collide at state s, if, in s, the value of t's program counter is pointing at a statement of one of the following procedures: LesOP, TryCollision, FinishCollision. Otherwise, we say that op is not trying to collide at s.* – **?**

Here the authors' intuitive concept of "trying to collide" is captured by the "Publish collision intent" quotient automaton state, as compared to the other states.

> *We next prove that operations can only collide with operations of the opposite type. First we need the following technical lemma. Lemma 5.2. Every colliding operation op is either active or passive, but not both.* – **?**

As discussed above, the bottom right 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. Furthermore, the bottom right layer shows that the colliding operations cannot be both active and passive.

> *Lemma 5.3. Operations can only collide with operations of the opposite type: an operation that performs a push can only collide with operations that perform a pop, and vice versa.* – ?

As discussed above, the bottom right 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.

> *Lemma 5.4. An operation terminates without modifying the central stack object, if and only if it collides with another operation.* – ?

This is captured by the bottom left layer, which (1) involves `return val` statements, avoiding the central stack and (2) is only reachable after a successful collision.

> *Lemma 5.5. For every thread p and in any state s, if p is not trying to collide in s, then it holds in s that the element corresponding to p in the location array is NULL.* – ?

Captured by the initial conditions and the (only possible) paths through the automaton.

> *Lemma 5.6. Let op be a push operation by some thread p; if location[p] ≠ NULL, then op is trying to push the value* `location[p]->cell.pdata`. – ?

Captured by the initial conditions and the (only possible) paths through the automaton.

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

As discussed above, the bottom right 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.

> *Lemma 5.8. Every active collider op1 collides with exactly one passive collider.* – ?

As discussed above, the bottom right 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.

> *Lemma 5.9. Every colliding operation op participates in exactly one collision with an operation of the opposite type.* – ?

As discussed above, the bottom right 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.

> *We now prove that, when colliding, opposite operations exchange values in a proper way. Lemma 5.10. If a pop operation collides, it obtains the value of the single push operation it collided with. [Lemma 5.11 analogous for push-pop.]* – ?

As discussed above, the bottom right 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.

> *We can now finally prove that our algorithm has correct set semantics. Theorem 5.12. The elimination-backoff stack has correct set semantics.* – ?

As discussed above, separately proving Set semantics is unnecessary.

Linearizability.

> *we choose the following linearization points for all operations, except for passive-colliders: Lines T4, C2 (for a push operation), Lines T10, T14, C7 (for a pop operation)* – ?

The authors give linearization points 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 are: (1) the bold successful CAS in the bottom right layer in the quotient automaton, and (2) the subsequent automaton transition where a corresponding

passive thread finds it has been collided with. 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, up to commutativity.

> *For a passive-collider operation, we set the linearization point to be at the time of linearization of the matching active-collider operation, and the push colliding-operation is linearized before the pop colliding-operation.* – **?**

Same as above.

> *Each push or pop operation consists of a while loop that repeatedly attempts to complete the operation. An iteration is successful if its attempt succeeds, in which case the operation returns at that iteration; otherwise, another iteration is performed . Each completed operation has exactly one successful attempt (its last attempt), and the linearization of the operation occurs in that attempt. In other words, the operations are linearized in the aforementioned lineanirazation points only in case of a successful CAS, which can only be performed in the last iteration of the while loop.* – **?**

Same as above.

> *To prove that the aforementioned lines are correct linearization points of our algorithm, we need to prove that these are correct linearization points for the two types of operations: operations that complete by modifying the central stack object, and operations that exchange values through collisions.* – **?**

Same as above.

> *Lemma 5.13. For operations that do not collide, we can choose the following linearization points: Line T4 (for a push operation). Line T10 (in case of empty stack) or line T14 (for a pop operation)* – **?**

Follows from the quotient automaton for the Treiber central stack.

> *We still have to prove that the linearization points for collider-operations are consistent, both with one another, and with non-colliding operations. We need the following technical lemma, whose proof is omitted for lack of space. Lemma 5.14. Let op1, op2, be a colliding operations-pair, and assume w.l.o.g. that op1 is the active-collider and op2 is the passive collider, then the linearization point of op1 (as defined above) is within the time interval of op2.* – **?**

Same as above.

> *Lemma 5.15. The following are legal linearization points for collider-operations. • An active-collider, op1, is linearized at either line C2 (in case of a push operation) or at line C7 (in case of a pop operation). • A passive-collider, op2, is linearized at the linearization time of the active-collider it collided with. If op2 is a push operation, it is linearized immediately before op1, otherwise it is linearized immediately after op1.* – **?**

Same as above.

Summary. 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 linearization point for that operation as well as the corresponding matched operation (order with the push before the pop). Specifically, 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.

Many of the lemmas and reasoning in the **?** proof are used to set up a bijection between active and passive threads. The quotient instead simplifies the proof through the serialized representative executions. The quotient similarly simplifies the other logistics of threads preparing/completing in the other quotient automaton states.

N.3 Michael-Scott Queue

The layer quotient for the MSQ is given in Apx. G. We will refer to the layers defined there.

> *We now review all the steps in detail. An enqueuer creates a new node with the new value to be enqueued (Line 10), reads tail, and finds the node that appears to be last (Lines 12–13). To verify that node is indeed last, it checks whether that node has a successor (Line 15). If so, the thread attempts to append the new node by calling compareAndSet() (Line 16). (A compareAndSet() is required because other threads may be trying the same thing.)* – Herlihy and Shavit [2008b]

The above scenario involves a single successful enqueuer and unboundedly many other enqueuers attempting. This scenario is captured by the *Enqueue Succeed Layer*, and the automaton transitions shown in Fig. 1.

> *If the compareAndSet() succeeds, the thread uses a second compareAndSet() to advance tail to the new node (Line 17). Even if this second compareAndSet() call fails, the thread can still return successfully because, as we will see, the call fails only if some other thread "helped" it by advancing tail.* – Herlihy and Shavit [2008b]

The above scenario corresponds to the *Advancer Succeed Layer*, where some advancer succeeds.

> *If the tail node has a successor (Line 20), then the method tries to "help" other threads by advancing tail to refer directly to the successor (Line 21) before trying again to insert its own node.* – Herlihy and Shavit [2008b]

The above scenario corresponds to the *Advancer Succeed Layer*, and the fact that "trying again to insert" occurs in a subsequent layer.

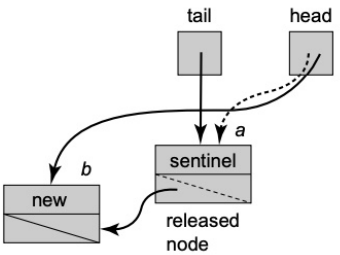
> *This enq() is total, meaning that it never waits for a dequeuer. A successful enq() is linearized at the instant where the executing thread (or a concurrent helping thread) calls compareAndSet() to redirect the tail field to the new node at Line 21.* – Herlihy and Shavit [2008b]

This linearization point occurrence is preserved in the layer quotient abstraction, at the point where the tail is advanced.

> *The deq() method is similar to its total counterpart from the UnboundedQueue. If the queue is nonempty, the dequeuer calls compareAndSet() to change head from the sentinel node to its successor, making the successor the new sentinel node. The deq() method makes sure that the queue is not empty in the same way as before: by checking that the next field of the head node is not null.* – Herlihy and Shavit [2008b]

This scenario is captured by the *Dequeue Succeed Layer* (when the queue is non-empty) and by *Read Only Layer 1* (where dequeue returns because the queue was empty).

Regarding ADT states, the correctness argument mentions two aspects: (1) whether the queue was empty and (2) whether the tail pointer was lagged. This is captured in the automaton representation in Fig. 1, where the states capture both 1 and 2.



> *There is, however, a subtle issue in the lock-free case, depicted [above]: before advancing head one must make sure that tail is not left referring to the sentinel node which is about to be removed from the queue. To avoid this problem we add a test: if head equals tail (Line 31) and the (sentinel) node they refer to has a non-null next field (Line 32), then the tail is deemed to be lagging behind. As in the enq() method, deq() then attempts to help make tail consistent by swinging it to the sentinel node's successor (Line 35), and only then updates head to remove the sentinel (Line 38). As in the partial queue, the value is read from the successor of the sentinel node (Line 37). If this method returns a value, then its linearization point occurs when it completes a successful compareAndSet() call at Line 38, and otherwise it is linearized at Line 33.* – Herlihy and Shavit [2008b]

There are multiple layers discussed above. First, there is an *Advancer Succeed Layer* as part of a dequeue. Second, a *Dequeue Succeed Layer* (or *Read Only Layer 1*) may occur, but only after ("only then") the *Advancer Succeed Layer*. This scenario is focused on the refers to q_3 in the quotient automaton, where `Q.tail=Q.head` and yet `Q.tail->next≠null`. The automaton helps illuminate this case because the states and arcs require one to consider all possible cases, which layers are enabled, and where the arcs land after the layer.

Summary. See Sec. 6.1.

N.4 SLS Queue

We review the correctness argument presented by the original authors [Scherer III et al. 2006], quoting their prose and discussing how those statements correspond to our quotient proof methodology with layer expressions.

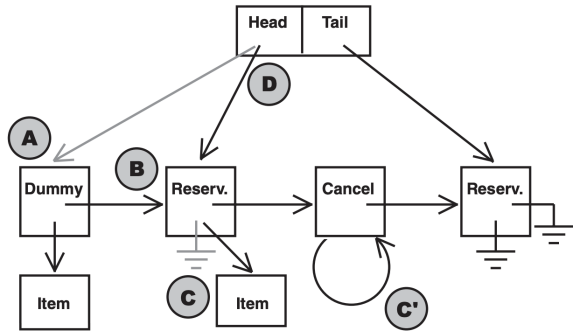
> *The reservation linearization point for this code path occurs at line 10 when we successfully insert our offering into the queue* – Scherer III et al. [2006]

First, this prose indicates that 3t is a linearization point. The write of 3t is atomic and so this line number has a corresponding location in the layer expression, which is this same linearization point. Second, this prose identifies a layer as an important state change: inserting an offer node into the queue. This is the EAIN layer in our decomposition. Third, the prose describes what kind of data change is important: the tail changing to non-null, a distinction we make in the states of our layer automaton.

> *"A successful followup linearization point occurs when we notice at line 13 that our data has been taken.* – Scherer III et al. [2006]

Similarly here this linearization point appears in a layer where a dequeue mutates the state, and local path (3") is feasible. This prose also identifies important state change: from an item to null.

> *The other case occurs when the queue consists of reservations (requests for data), and is depicted [below].* – Scherer III et al. [2006]



> *In this case, after originally reading the head node (step A), we read its successor (line 21/step B) and verify consistency (line 22). Then, we attempt to supply our data to the head-most reservation (line 25/C). If this succeeds, we dequeue the former dummy node (26/D) and return* – Scherer III et al. [2006]

This prose again indicates important state changes, which are reflected as distinct states (and transitions between them) in our layer automata: whether head-most reservation has data supplied and whether the head dummy node needs to be advanced.

> *If it fails, we need to go to the next reservation, so we dequeue the old dummy node anyway (28) and retry the entire operation (32, 05).* – Scherer III et al. [2006]

This is a description of the failure path 5f · (6bt + (6bf)) and that interference (implicitly) caused by a concurrent cas from 5t.

> *The reservation linearization point for this code path occurs when we successfully supply data to a waiting consumer at line 25; the followup linearization point occurs immediately thereafter.* – Scherer III et al. [2006]

Again, this prose indicates the important state transition at 5t, replacing a null with an item (as seen in the states of our layer automaton), and corresponding automaton transition for layer EFHR.

Summary. A summary is given in Sec. 6.2.

N.5 Herlihy-Wing Queue

We now examine the author's proof of this object. As discussed in Sec. 6.5, the quotient can be abstracted as: $(\texttt{deqF}^* \cdot (\texttt{enqI})^+ \cdot \texttt{enqW}^* \cdot \texttt{deqT}^*)^*$. A proof of correctness is given in Appendix II of Herlihy and Wing [1990]. A key challenge of this object is that linearization points are non-fixed.

> *An Enq execution occurs in two distinct 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 an execution 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 linearization points (in this and other objects), the authors introduce a proof methodology based on tracking all possible linearizations that could happen in the future:

> *For each linearized value, it is sometimes useful to keep track of which invocations were completed in the linearization that yielded that value, and what their responses were. A possibility for a history H is a triple (v, P, R), where v is a linearized value of H, P is the subset of pending invocations in H not completed when forming the linearization that yielded u, and R is the set of responses appended to H to form u.* – Appendix I of Herlihy and Wing [1990]

This is a rather general method for linearizability. The quotient, however, 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.

Importantly, while the Appendix I of Herlihy and Wing [1990] methdology maintains a history to allow for all possible linearization orders, quotient-based reasoning instead involves representative executions (those that are accepted by the regular expression) with *fixed* linearization orders and all other executions are equivalent to one such representative execution upto commutativity.