# Between Linearizability and Quiescent Consistency[⋆]
## Quantitative Quiescent Consistency

Radha Jagadeesan and James Riely

DePaul University

**Abstract** Linearizability is the de facto correctness criterion for concurrent data structures. Unfortunately, linearizability imposes a performance penalty which scales linearly in the number of contending threads. Quiescent consistency is an alternative criterion which guarantees that a concurrent data structure behaves correctly when accessed sequentially. Yet quiescent consistency says very little about executions that have any contention.

We define quantitative quiescent consistency (QQC), a relaxation of linearizability where the degree of relaxation is proportional to degree of contention. When quiescent, no relaxation is allowed, and therefore QQC refines quiescent consistency, unlike other proposed relaxations of linearizability. We show that high performance counters and stacks designed to satisfy quiescent consistency continue to satisfy QQC. The precise assumptions under which QQC holds provides fresh insight on these structures. To demonstrate the robustness of QQC, we provide three natural characterizations and prove compositionality.

## 1 Introduction

This paper defines *Quantitative Quiescent Consistency (QQC)* as a criterion that lies between linearizability [9] and quiescent consistency [3, 10]. The following example should give some intuition about these criteria.

*Example 1.1.* Consider a counter object with a single `getAndIncrement` method. The counter's sequential behavior can be defined as a set of strings such as $[^{+}\ ]^{+}_{0}\ \{^{+}\ \}^{+}_{1}\ (^{+}\ )^{+}_{2}$ where $[^{+}$ denotes an invocation (or call) of the method and $]^{+}_{i}$ denotes the response (or return) with value $i$. Suppose each invocation is initiated by a different thread.

A concurrent execution may have overlapping method invocations. For example, in $(^{+}\ [^{+}\ ]^{+}_{0}\ \{^{+}\ \}^{+}_{1}\ )^{+}_{2}$ the execution of $(^{+}\ )^{+}_{2}$ overlaps with both $[^{+}\ ]^{+}_{0}$ and $\{^{+}\ \}^{+}_{1}$, whereas $[^{+}\ ]^{+}_{0}$ finishes executing before $\{^{+}\ \}^{+}_{1}$ begins. Consider the following four executions.

$$(^{+}\ [^{+}\ ]^{+}_{0}\ \{^{+}\ \}^{+}_{1}\ )^{+}_{2} \qquad (^{+}\ \{^{+}\ \}^{+}_{1}\ [^{+}\ ]^{+}_{0}\ )^{+}_{2} \qquad [^{+}\ (^{+}\ )^{+}_{2}\ \{^{+}\ \}^{+}_{1}\ ]^{+}_{0} \qquad [^{+}\ (^{+}\ )^{+}_{2}\ ]^{+}_{0}\ \{^{+}\ \}^{+}_{1}$$

*Linearizability* states roughly that *every* response-to-invocation order in a concurrent execution must be consistent with the sequential specification. Thus, the first execution is linearizable, since the response $]^{+}_{0}$ precedes the invocation $\{^{+}$ in the specification. However, none of the other executions is linearizable. For example, $\}^{+}_{1}$ precedes $[^{+}$ in the second execution, but $\}^{+}_{1}$ does not precedes $[^{+}$ in the specification.

⋆ Research supported by NSF 0916741.

Linearizability can also be understood in terms the *linearization point* of a method execution, which must occur between the invocation and response. From this perspective, the first execution above is linearizable because we can find a sequence of linearization points that agrees with the specification; this requires only that the linearization point of $(^+\ )^+_2$ follow that of $\{^+\ \}^+_1$. No such sequence of linearization points exists for the two other executions.

*Quiescent consistency* is similar to linearizability, except that the response-to-invocation order must be respected only across a quiescent point, that is, a point with no open method calls. The first three executions above are quiescently consistent simply because there are no non-trivial quiescent points. The last execution fails to be quiescently consistent since the order from $)^+_2$ to $\{^+$ is not preserved in the specification.

We define *Quantitative Quiescent Consistency (QQC)* to require that the number of response-to-invocation pairs that are out-of-order at any point be bounded by the number of open calls at that point. We also give a *counting characterization* of QQC, which requires that if a response matches the $i^{th}$ method call in the specification, then it must be preceded by at least $i$ invocations.

The first two executions above are QQC; however, the last two are not. In the second execution, the open call to $(^+$ is "enough" to justify the return of $\{^+\ \}^+_1$ before $[^+\ ]^+_0$. However, in the third execution, the return of $(^+\ )^+_2$ before $\{^+\ \}^+_1$ cannot be justified only by the call to $[^+$; it is "too far off." Following the counting characterization sketched above, the third execution fails since $(^+\ )^+_2$ is the third method call in the specification trace, but $)^+_2$ is only preceded by two invocations: $[^+$ and $(^+$. □

Quiescent consistency is too coarse to be of much use in reasoning about concurrent executions. For example, a sequence of interlocking calls never reaches a quiescent point; therefore it is trivially quiescently consistent. This includes obviously correct executions, such as $[^+\ (^+\ ]^+_0\ [^+\ )^+_1\ (^+\ ]^+_2\ [^+\ )^+_3\ (^+\ ]^+_4\ [^+\ \cdots$, nearly correct executions, such as $[^+\ (^+\ ]^+_1\ [^+\ )^+_0\ (^+\ ]^+_3\ [^+\ )^+_2\ (^+\ ]^+_5\ [^+\ \cdots$, and also ridiculous executions, such as $[^+\ (^+\ ]^+_{1074}\ [^+\ )^+_{17}\ (^+\ ]^+_{2344}\ [^+\ )^+_3\ (^+\ \cdots$.

Linearizability has proven quite useful in reasoning about concurrent executions; however, it fundamentally constrains efficiency in a multicore setting: Dwork, Herlihy, and Waarts [5] show that if many threads concurrently access a linearizable counter, there must be either a location with high contention or an execution path that accesses many shared variables.

Shavit [11] argues that the performance penalty of linearizable data structures is increasingly unacceptable in the multi-core age. This observation has lead to a recent renewal of interest in nonlinearizable data structures. As a simple example, consider the following counter implementation: a simplified version of the counting networks of Aspnes, Herlihy, and Shavit [3].

```
class Counter<N:Int> {
   field b:[0..N-1] = 0;                  // 1 balancer
   field c:Int[]    = [0, 1, ..., N-1]; // N counters
   method getAndIncrement():Int {
      val i:[0..N-1];
      atomic { i = b; b++; }
      atomic { val v = c[i]; c[i] += N; return v; } } }
```

The $N$-`Counter` has two fields: a *balancer* `b` and an array `c` of $N$ integer counters. There are two atomic actions in the code: The first reads and updates the balancer, setting the local index variable `i`. The second reads and updates the `i`$^{th}$ counter. Although the balancer has high contention in our simplified implementation, the counters do not; balancers that avoid high contention are described in [3].

*Example 1.2.* The $N$-`Counter` behaves like a sequential counter if calls to `getAndIncrement` are sequentialized. To see this, consider a 2-`Counter`, with initial state $\langle \mathtt{b} = 0, \mathtt{c} = [[0], [1]]\rangle$. In a series of sequential calls, the state progresses as follows, where we show the execution of the first atomic with the invocation and the second atomic with the response. The execution $[^+\ ]^+_0\ \{^+\ \}^+_1\ (^+\ )^+_2$ can be elaborated as follows.

$$\begin{aligned}
\langle \mathtt{b} = 0, \mathtt{c} = [[0], [1]]\rangle &\xrightarrow{[^+} \langle \mathtt{b} = 1, \mathtt{c} = [[0], [1]]\rangle \xrightarrow{]^+_0} \langle \mathtt{b} = 1, \mathtt{c} = [[2], [1]]\rangle \\
&\xrightarrow{\{^+} \langle \mathtt{b} = 0, \mathtt{c} = [[2], [1]]\rangle \xrightarrow{\}^+_1} \langle \mathtt{b} = 0, \mathtt{c} = [[2], [3]]\rangle \\
&\xrightarrow{(^+} \langle \mathtt{b} = 1, \mathtt{c} = [[2], [3]]\rangle \xrightarrow{)^+_2} \langle \mathtt{b} = 1, \mathtt{c} = [[4], [3]]\rangle
\end{aligned}$$

When there is concurrent access, the 2-`Counter` allows nonlinearizable executions, such as $(^+\ \{^+\ \}^+_1\ [^+\ ]^+_0\ )^+_2$ .

$$\begin{aligned}
\langle \mathtt{b} = 0, \mathtt{c} = [[0], [1]]\rangle &\xrightarrow{(^+} \langle \mathtt{b} = 1, \mathtt{c} = [[0], [1]]\rangle \\
&\xrightarrow{\{^+} \langle \mathtt{b} = 0, \mathtt{c} = [[0], [1]]\rangle \xrightarrow{\}^+_1} \langle \mathtt{b} = 0, \mathtt{c} = [[0], [3]]\rangle \\
&\xrightarrow{[^+} \langle \mathtt{b} = 1, \mathtt{c} = [[0], [3]]\rangle \xrightarrow{]^+_0} \langle \mathtt{b} = 1, \mathtt{c} = [[2], [3]]\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \xrightarrow{)^+_2} \langle \mathtt{b} = 1, \mathtt{c} = [[4], [3]]\rangle
\end{aligned}$$

With a sequence of interlocking calls, it is also possible for the $N$-`Counter` to execute as $[^+\ (^+\ ]^+_1\ [^+\ )^+_0\ (^+\ ]^+_3\ [^+\ )^+_2\ (^+\ ]^+_5\ [^+\ \cdots$ , producing an infinite sequence of values that are just slightly out of order. Using the results of this paper, one can conclude that with a maximum of two open calls, the value returned by `getAndIncrement` will be "off" by no more than 2, but this does not follow from quiescent consistency. □

Our results are related to those of [2–4, 13]. In particular, Aspnes, Herlihy, and Shavit [3] prove that in any *quiescent* state (with no call that has not returned), such a counter has a "step-property", indicating the shape of `c`. Between $\}^+_1$ and $]^+_0$ in the second displayed execution of Example 1.2, the states with $\mathtt{c} = [[0], [3]]$ do *not* have the step property, since the two adjacent counters differ by more than 1.

Aspnes, Herlihy, and Shavit imply that the step property is related to quiescent consistency, but they do not formally state this. Indeed, they do not provide a formal definition of quiescent consistency. It appears that they have in mind is something like the following: An execution is *weakly quiescent consistent* if any subsequence of *sequential* calls (single calls separated by quiescent points) is a subtrace of a specification trace.

The situation is delicate: Although the increment-only counters of [3] are quiescently consistent in the sense we defined in Example 1.1 (indeed, they are QQC), the increment-decrement counters of [2, 4, 13] are only *weakly* quiescent consistent. Indeed, the theorems proven in [13] state only that, at a quiescent point, a variant of the step property holds. It states nothing about the actual values read from the individual counters. Instead, we expect that a quiescently consistent execution should be a permutation of *some* specification trace, even if it has no nontrivial quiescent points.

*Example 1.3.* Consider an extension of the `2-Counter` with `decrementAndGet`.

```
method decrementAndGet():Int {
   val i:[0..N-1];
   atomic { i = b-1; b--; }
   atomic { c[i] -= N; return c[i]; } }
```

The execution $[^{+}\ \{^{+}\ (^{-}\ <^{-}\ >^{-}_{-2}\ ]^{+}_{-2}\ \}^{+}_{1}\ )^{-}_{1}$ is possible, although this is not a permutation of any specification trace. The execution proceeds as follows.

$$\begin{aligned}
\langle \mathtt{b}=0, \mathtt{c}=[[0],[1]]\rangle &\xrightarrow{[^{+}} \langle \mathtt{b}=1, \mathtt{c}=[[0],[1]]\rangle \xrightarrow{\{^{+}} \langle \mathtt{b}=0, \mathtt{c}=[[0],[1]]\rangle\\
&\xrightarrow{(^{-}} \langle \mathtt{b}=1, \mathtt{c}=[[0],[1]]\rangle \xrightarrow{<^{-}} \langle \mathtt{b}=0, \mathtt{c}=[[0],[1]]\rangle\\
&\xrightarrow{>^{-}_{-2}} \langle \mathtt{b}=0, \mathtt{c}=[[-2],[1]]\rangle \xrightarrow{]^{+}_{-2}} \langle \mathtt{b}=0, \mathtt{c}=[[0],[1]]\rangle\\
&\xrightarrow{\}^{+}_{1}} \langle \mathtt{b}=0, \mathtt{c}=[[0],[3]]\rangle \xrightarrow{)^{-}_{1}} \langle \mathtt{b}=0, \mathtt{c}=[[0],[1]]\rangle \qquad \Box
\end{aligned}$$

It is important to emphasize that this increment-decrement counter is not even quiescently consistent. There is no hope that it could satisfy any stronger criterion.

Of course counters are not the only data structures of interest. In this paper we treat concurrent stacks in detail. We define a simplified $N$-`Stack` below; the full, tree-based data structure is defined in Shavit and Touitou [13] and summarized in section 6.

```
class Stack<N:Int> {
   field b:[0..N-1] = 0;                      // 1 balancer
   field s:Stack[]  = [[], [], ..., []]; // N stacks of values
   method push(x:Object):Unit {
      val i:[0..N-1];
      atomic { i = b; b++; }
      atomic { val v = s[i].push(x); return v; } }
   method pop():Object {
      val i:[0..N-1];
      atomic { i = b-1; b--; }
      atomic { val v = s[i].pop(); return v; } } }
```

The trace given in Example 1.3 for the increment-decrement counter is also a trace of the stack, where we interpret `+` as `push`, `-` as `pop`, and a negative return value as stack underflow. Whereas this is a nonsense execution for a counter, it is a linearizable execution of a stack: simply choose the linearization point of the pops before the pushes. Nonetheless, the $N$-`Stack` is only *weakly* quiescent consistent in general.

*Example 1.4.* The $N$-`Stack` generates the execution $[^{+}_{a}\ ]^{+}\ (^{+}_{b}\ )^{+}\ \{^{+}_{c}\ <^{-}\ >^{-}_{a}\ \}^{+}$ as follows.

$$\begin{aligned}
\langle \mathtt{b}=0, \mathtt{s}=[[\,],[\,]]\rangle &\xrightarrow{[^{+}_{a}} \langle \mathtt{b}=1, \mathtt{s}=[[\,],[\,]]\rangle \xrightarrow{]^{+}} \langle \mathtt{b}=1, \mathtt{s}=[[a],[\,]]\rangle\\
&\xrightarrow{(^{+}_{b}} \langle \mathtt{b}=0, \mathtt{s}=[[a],[\,]]\rangle \xrightarrow{)^{+}} \langle \mathtt{b}=0, \mathtt{s}=[[a],[b]]\rangle\\
&\xrightarrow{\{^{+}_{c}} \langle \mathtt{b}=1, \mathtt{s}=[[a],[b]]\rangle\\
&\xrightarrow{<^{-}} \langle \mathtt{b}=0, \mathtt{s}=[[a],[b]]\rangle \xrightarrow{>^{-}_{a}} \langle \mathtt{b}=0, \mathtt{s}=[[\,],[b]]\rangle\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \xrightarrow{\}^{+}} \langle \mathtt{b}=0, \mathtt{s}=[[c],[b]]\rangle
\end{aligned}$$

However, this specification is not quiescently consistent with any stack execution: There is a quiescent point after each of the first two pushes; therefore it is impossible to pop $a$ before $b$. This execution is possible even when there are several pushes beforehand. □

In the case of the $N$-`Stack`, a simple *local* constraint can be imposed in order to establish quiescent consistency: intuitively, we require that no pop *overtakes* a push on the same stack `s[i]`. In section 6 we show that the stack is actually *QQC* under this constraint, and therefore quiescently consistent. (We have not found a local constraint under which the increment-decrement counter is quiescently consistent; we believe that it may be achievable with a global toggle that determines how to resolve the races at each point, but this, of course, defeats the point.)

The correctness result that we prove for elimination-tree stacks in section 6 is much stronger than that of Shavit and Touitou [13], who only prove *weak* quiescent consistency. The same is holds for increment-only counters [3], although in this case, we have elided the proofs: The proofs for the tree-based increment-only counter follow the structure proofs for the elimination-tree stacks.

The preliminary version of Shavit and Touitou's paper [12] suggests an upcoming definition *ε-linearizability*, "a variant of linearizability that captures the notion of 'almostness' by allowing a certain fraction of concurrent operations to be out-of-order." Since the details did not make it into the final version of the paper [13], it is unclear whether the "fraction of concurrent operations" is meant to vary depending on the amount of concurrency available in execution at any given moment, or if the "fraction" is fixed at the outset. If it is meant to vary, then it would be very similar to QQC.

This thread was picked up by Afek, Korland, and Yanovsky [1] and improved by Henzinger, Kirsch, Payer, Sezgin, and Sokolova [8]. As defined in [8], the idea is to define a cost metric on relaxations of strings and to bound the relaxation cost for the specification trace that matches an execution. This approach has been used to validate several novel concurrent data structures [1, 6].

Unlike QQC, the metrics in these papers do not depend on the available concurrency in the execution. In fact, with the exception of the increment-only counter validated in [1], all of the concurrent data structures of [1, 6] intentionally violate quiescent consistency. As such, their work is orthogonal to the approach we pursue here, which specifically refines quiescent consistency.

The primary difference between QQC and the relaxation-based approach of [1, 8] can be stated as follows: with QQC, the maximal degradation depends upon the amount of concurrent access, whereas in the relaxation-based approach it does not. Thus, QQC "degrades gracefully" as concurrency increases. In particular, a QQC data structure that is accessed sequentially will exactly obey the sequential specification, whereas a data structure validated against the relaxation-based approach may not.

In the rest of the paper, we formalize QQC and study its properties. The heart of the paper is section 5, which defines QQC and establishes its properties. The impatient reader can safely skim up to that section, referring back as necessary.

Our contributions are as follows.

– We define linearizability (section 3), quiescent consistency (section 4) and QQC (section 5) in terms of partial orders over events with duration. The formalities of the model are described in section 2. As in Example 1.1, the definitions are given in terms of the order between a response and a subsequent invocation.

- We provide an alternative characterization of QQC in terms of the number of invocations that precede a response, as well as a similar characterization of linearizability.
- We provide an alternative characterization of QQC in terms of a proxy that controls access to the underlying sequential data structure. The proxy adds a form of *speculation* to the flat combining technique of Hendler, Incze, Shavit, and Tzafrir [7]. We show that the operational semantics is sound and complete for QQC; that is, it generates *all and only* traces that are QQC.
- Like linearizability and quiescent consistency [10], QQC is non-blocking and compositional. Like quiescent consistency and unlike linearizability, a QQC execution may not respect program order. We provide a proof for compositionality.
- We show that QQC is useful for reasoning about data structures in the literature. In section 6, we prove that the elimination tree stacks of Shavit and Touitou [13] are QQC, as long as no pop overtakes a push on the same stack.

## 2 Model

The semantics of a concurrent program is given as a process. A *process* is a set of traces. A *trace* is a finite, polarized LPO (labelled partial order). Formally, we define traces to be finite sets of named *events*. The event names are the carrier set for the LPO, and the order is embedded in the events themselves using name sets.

### 2.1 Events

An event is a quadruple, consisting of a polarity, a label, a name (identifying a node the partial order) and a set of names (identifying the preceding nodes in the partial order). As a standard example, the reader may want to consider labels generated by the grammar $\ell ::= \mathsf{call}\,\tau\,o\,f\,w \mid \mathsf{ret}\,\tau\,o\,f\,w$ where $\tau$ is a thread identifier, $o$ is an object name, $f$ is a function name, and $w$ is the actual parameter or return value.

Let $a, b \in \mathit{Name}$ range over names and $A, B \subseteq \mathit{Name}$ range over finite sets of names. And let $\ell \in \mathit{Label}$ range over labels (with some interpretation in the application domain). Then events are defined as follows[1].

$$u, v ::= \langle ?\ell\rangle^a_A \mid \langle b\ell\rangle^a_A$$

Under our standard example, we would expect events to come in pairs of the form $\langle ?\mathsf{call}\,\tau\,o\,f\,w\rangle^a_A$ and $\langle a\,\mathsf{ret}\,\tau\,o\,f\,w'\rangle^b_B$, where $w$ is the actual parameter and $w'$ is the returned value. The appearance of $a$ in the return event indicates that this event closes the open call named $a$.

Three of the components in an event can be retrieved simply. We use the following functions: $\mathsf{label}(\langle ?\ell\rangle^a_A) \triangleq \ell$, $\mathsf{id}(\langle ?\ell\rangle^a_A) \triangleq a$ and $\mathsf{before}(\langle ?\ell\rangle^a_A) \triangleq A$. For the remaining component, we define both the functions $\mathsf{pol}$ and $\mathsf{brak}$. Let $\rho \in \{?, !\}$ range over the polarities for input ($?$) and output ($!$) and let $\mathsf{none}$ be a reserved name.

$$\mathsf{pol}(u) \triangleq \begin{cases} ? & \text{if } u = \langle ?\ell\rangle^a_A \\ ! & \text{if } u = \langle b\ell\rangle^a_A \end{cases} \qquad \mathsf{brak}(u) \triangleq \begin{cases} \mathsf{none} & \text{if } u = \langle ?\ell\rangle^a_A \\ b & \text{if } u = \langle b\ell\rangle^a_A \end{cases}$$

[1] In this paper, we consider the simple case of non-interacting composition. This allows us to ignore the internal polarity which arise from the interaction of input and output.

Because the standard example is so familiar, we will consider invocation/call/input/$\texttt{?}$ to be synonymous, and likewise response/return/output/$\texttt{!}$.

We sometimes use superscripts on name metavariables, such as $a^{!}$ and $a^{?}$. Any name bound to $a^{!}$ must be associated with an output event, and likewise for input events. The superscript makes these distinct metavariables. Thus we have $a^{!} \neq a^{?}$.

Turning to the order between events, we write $u \Rightarrow v$ to indicate that $u$ precedes $v$: $(u \Rightarrow v) \triangleq \mathsf{id}(u) \in \mathsf{before}(v)$.

## 2.2 Traces

We use $p$–$t$ to range over *event sets* (finite sets of events). Define $\mathsf{ids}(s) \triangleq \{\mathsf{id}(u) \mid u \in s\}$ and let $a \in s$ be shorthand for $a \in \mathsf{ids}(s)$.

Given an event set $s$ and name set $A$, define *indexing* as $s[A] \triangleq \{u \in s \mid \mathsf{id}(u) \in A\}$. Thus $s[\mathsf{ids}(s)] = s$. If event names are unique, this generates the partial function $s[a]$ for single names: if $s[\{a\}] = \emptyset$ then $s[a]$ is undefined; if $s[\{a\}] = \{u\}$ then $s[a] \triangleq u$. Indexing provides a natural way to lift ordering relations from events to names: $(a \Rightarrow_s b) \triangleq (s[a] \Rightarrow s[b])$. Let be $\Rrightarrow_s$ the reflexive closure of $\Rightarrow_s$.

An event set $s$ is a *trace* if it satisfies the following, $\forall u, v \in s$.

(1) event names are unique: if $\mathsf{id}(u) = \mathsf{id}(v)$ then $u = v$
(2) before okay: $\forall a \in \mathsf{before}(u).\ \exists w \in s.\ a = \mathsf{id}(w)$
(3) brak okay: if $\mathsf{pol}(u) = \texttt{!}$ then $\mathsf{brak}(u) \in \mathsf{before}(u)$ and $\mathsf{pol}(s[\mathsf{brak}(u)]) = \texttt{?}$
(4) input acquires control: if $a \Rightarrow_s b^{?}$ then $\exists c^{!}.\ a \Rrightarrow_s c^{!} \Rightarrow_s b^{?}$
(5) output releases control: if $a^{!} \Rightarrow_s b$ then $\exists c^{?}.\ a^{!} \Rrightarrow_s c^{!} \Rightarrow_s b$
(6) $\Rightarrow_s$ defines a strict partial order (irreflexive, antisymmetric and transitive)

A trace $s$ is *operational* if $\forall a^{?}, b^{!} \in s$. either $a^{?} \Rightarrow_s b^{!}$ or $b^{!} \Rightarrow_s a^{?}$.
A trace $s$ is *sequential* if $\forall a, b \in s$. either $a \Rightarrow_s b$ or $b \Rightarrow_s a$.

Our model can be viewed as a labelled partial order enriched with polarity and bracketing. Most significant here are requirements (4) and (5) in the definition of a trace. One immediate consequence is that input events cannot be related to other input events unless there is an intervening output event, and similarly for the dual case.

Consider two bracketed event sequences $\texttt{[]}$ and $\texttt{()}$. As indicated by condition (3) in the definition of traces, the open brackets must be $\texttt{?}$ events. There are six possible relations among the events. Three of these are familiar: it could be that $\texttt{[]}$ precedes $\texttt{()}$, or that $\texttt{()}$ precedes $\texttt{[]}$ or that they are concurrent.

$$\begin{tikzcd} \texttt{[} \arrow[r] & \texttt{]} \arrow[r] & \texttt{(} \arrow[r] & \texttt{)} \end{tikzcd} \qquad \begin{tikzcd} \texttt{(} \arrow[r] & \texttt{)} \arrow[r] & \texttt{[} \arrow[r] & \texttt{]} \end{tikzcd} \qquad \begin{tikzcd} \texttt{[} \arrow[r] \arrow[dr] & \texttt{]} \\ \texttt{(} \arrow[r] \arrow[ur] & \texttt{)} \end{tikzcd}$$

All of these traces are fully specified in the sense that every $\texttt{?}$ is ordered with respect to every $\texttt{!}$, and dually every $\texttt{!}$ is ordered with respect to every $\texttt{?}$. We call such traces *operational* in that they correspond to traces generated by an interleaving semantics. In addition, the first two traces are *sequential*, since there is a total order on the events.

Note that in any sequential trace, the initial event must be an input; this follows from properties (2) and (3) in the definition of traces.

There is a homomorphism from strings of bracketed labels to operational traces: each input in the string is ordered with respect to each output that follows it in the string, and dually. If we narrow attention to sequential traces, this is an isomorphism. For example, we can write the first two traces above as the strings `[]()` and `()[]`, respectively. The last trace above can be written as any interleaving in `[]` $|||$ `()` that orders the inputs before both outputs; these are `[(])`, `([])`, `[()]`, and `([)]`. We use this notation when giving examples of operational traces, as in the introduction.

As a consequence of the homomorphism, we can use string notation on operational traces without ambiguity. Specifically, let $st$ represent the concatenation of $s$ and $t$ and $s \,|||\, t$ represent the set of their interleavings, with renaming as necessary to avoid collisions between the names of $s$ and $t$.

Our model also allows underspecification of the relationship.

$$\begin{tikzcd} \texttt{[} \arrow[r] \arrow[dr] & \texttt{]} \\ \texttt{(} \arrow[r] & \texttt{)} \end{tikzcd} \qquad \begin{tikzcd} \texttt{[} \arrow[r] & \texttt{]} \\ \texttt{(} \arrow[r] \arrow[ur] & \texttt{)} \end{tikzcd} \qquad \begin{tikzcd} \texttt{[} \arrow[r] & \texttt{]} \\ \texttt{(} \arrow[r] & \texttt{)} \end{tikzcd}$$

The leftmost of these says only that `()` cannot precede `[]`. Said positively, either `[]` precedes `()`, or they are concurrent. The rightmost of these places no constraints on the relative order of `[]` and `()`.

Operational traces can be seen as having a global notion of time: everyone agrees what happened before what. The constraints between pairs of inputs and pairs of output simply indicate the limits of observability: it is impossible to tell which of two calls happened first. In this light, one may view an underspecified trace as a representative for the set of operational traces that can be derived by augmenting the partial order. We take this viewpoint in our compositionality result, which is stated only for operational traces.

We define several notations for event sets and traces.

As noted above, for operational traces $s$ and $t$ we use string notation for concatenation ($st$) and the set of interleavings ($s \,|||\, t$).

A *renaming* of a trace is identical to the original trace up to a bijection on names[2]. We write $=_\alpha$ for equivalence up to renaming.

A *permutation* of a trace contains events with the same names, labels and polarities, but may differ in ordering. Permutation does *not* allow renaming; the names to pick out the witnessing bijection. We write $=_\pi$ for equivalence up to permutation[3].

A *prefix* is a down-closed subtrace[4]. We write $t \leq_{\mathsf{pre}} s$ or $s \geq_{\mathsf{pre}} t$ to indicate that $t$ is a prefix of $s$, and $\downarrow_s a$ for the smallest down-closed subset of $s$ that includes $a$.

[2] $(s =_\alpha t)$ is defined to mean that there exists a bijection $\alpha$ on names such that (1) $\mathsf{ids}(s) = \alpha$ $(\mathsf{ids}(t))$, and (2) $\forall a \in \mathsf{ids}(s).\ s[a] = t[\alpha(a)]$. (In the first condition, we have used the obvious homomorphic extension of $\alpha$ to sets of names.)

[3] Let $(u =^{\mathsf{label}}_{\mathsf{brak}} v) \triangleq (\mathsf{label}(u) = \mathsf{label}(v)) \wedge (\mathsf{brak}(u) = \mathsf{brak}(v))$.
Then define $(s =_\pi t) \triangleq (\mathsf{ids}(s) = \mathsf{ids}(t)) \wedge (\forall a \in s.\ s[a] =^{\mathsf{label}}_{\mathsf{brak}} t[a])$.

[4] Trace $t$ is a *prefix* of trace $s$ if $\forall a, b \in s$. if $a \in t$ and $b \Rightarrow_s a$ then $b \in t$.

We treat traces both as sets of events and as partial orders. We use $(-)$ for set difference and $(\div)$ for partial order difference[5].

## 3 Linearizability

We give two characterizations of linearizability and prove compositionality.

In subsection 3.1, we give a characterization that looks at every way to *cut* a trace into prefix and suffix; linearizability requires that response-to-invocation order be respected across all cuts. This corresponds to characterization of QQC given in subsection 5.1. In the case of QQC, a certain number of invocations may be ignored, proportional to the number of calls that are both open across the cut and out of specification-order with respect to the response.

In subsection 3.2 we give a subset-based characterization, which requires that if a response matches the $i^{th}$ method call in the specification, then it must be preceded by at the first $i$ invocations of the specification. This corresponds to characterization of QQC given in subsection 5.2. In the case of QQC, it is sufficient that a response by the $i^{th}$ method be preceded by any $i$ invocations, not necessarily the first $i$ invocations of the specification.

The proof of compositionality in subsection 3.3 is provided as a warmup for the proof compositionality for QQC in subsection 5.4.

### 3.1 First characterization: response to invocation

Intuitively, linearizability requires that the response-to-invocation order in an execution be respected by a specification trace. To show that $s''$ is linearizable, it suffices to do the following

- Choose a specification trace $t$.
- Choose an extension $s'$ of $s''$ that closes the open calls in $s''$. Let $\mathsf{extensions}(s'')$ be the set of *extensions* of $s''$, such that (1) if $s' \in \mathsf{extensions}(s'')$ then $s''$ is a prefix of $s'$ ($s' \geq_{\mathsf{pre}} s''$), and (2) all of the new events in $s' - s''$ are ordered after all events of opposite polarity in $s''$[6] (that is, calls after returns and returns after calls).
- Choose a renaming $s =_{\alpha} s'$ such that $s =_{\pi} t$. Informally, this establishes that $s'$ is a permutation of $t$. Rather than carrying the permutation around in the definition, as usual in definitions of linearizability, we perform a renaming up front, once and

[5] An event set $t$ is *bracketed* if every output in $t$ has a matching input in $t$; that is $\forall u \in t.$ if $\mathsf{pol}(u) = \mathbf{!}$ then $\mathsf{brak}(u) \in \mathsf{ids}(t)$. A bracketed set may contain unmatched inputs, but not unmatched outputs.

For arbitrary event sets, we write $s - t$ for set difference. For trace $s$ and bracketed event set $t$, we write $s \div t$ for partial order difference. For example, consider the trace the sequential trace $s = \langle ?\ell_1 \rangle^{a}_{\emptyset} \langle a\,\ell_2 \rangle^{a'}_{\{a\}} \langle ?\ell_3 \rangle^{b}_{\{a,a'\}} \langle b\,\ell_4 \rangle^{b'}_{\{a,a',b\}} \langle ?\ell_5 \rangle^{c}_{\{a,a',b,b'\}}$ and let $t$ be the bracketed set $\{s[b], s[b']\}$. Then we have $s - t = \langle ?\ell_1 \rangle^{a}_{\emptyset} \langle a\,\ell_2 \rangle^{a'}_{\{a\}} \langle ?\ell_5 \rangle^{c}_{\{a,a',b,b'\}}$ and $s \div t = \langle ?\ell_1 \rangle^{a}_{\emptyset} \langle a\,\ell_2 \rangle^{a'}_{\{a\}} \langle ?\ell_5 \rangle^{c}_{\{a,a'\}}$.

[6] $$\begin{aligned}\mathsf{extensions}(p) \triangleq \{s \mid p \leq_{\mathsf{pre}} s &\wedge \forall a^{!} \in p.\, \forall b^{?} \in s - p.\ a^{!} \Rightarrow_p b^{?} \\ &\wedge \forall a^{?} \in p.\, \forall b^{!} \in s - p.\ a^{?} \Rightarrow_p b^{!}\}\end{aligned}$$

for all. The names are then witness to the permutation. This works nicely, since our traces are indexed by names. Typically, linearizability is defined over strings, indexed by integers, so this technique is not available.

– Show that for every response $a^!$ in $s$ and invocation $b^?$ in $s$, if $a^!$ precedes $b^?$ in $s$ ($a^! \Rightarrow_s b^?$), then the same must be true in $t$ ($a^! \Rightarrow_t b^?$).

Stated compactly, we have the following definition.

*Definition 3.1.* Trace $s''$ *linearizes* to $t$ if $\exists s' \in \mathsf{extensions}(s'').\ \exists s =_\alpha s'.\ s =_\pi t$ and

$$\forall a^! \in s.\ \forall b^? \in s.\ (a^! \Rightarrow_s b^?) \text{ implies } (a^! \Rightarrow_t b^?).$$

Trace set $S$ *linearizes* to $T$ if $\forall s'' \in S.\ \exists t \in T.\ s''$ linearizes to $t$. □

This definition differs from the traditional one in several small details, but is equivalent under reasonable assumptions. The differences are as follows.

– We do not require that specifications be sequential.
– We do not make requirements specific to threads. A thread is simply a totally ordered sequence of actions, with the result that every pair of invocations must be separated by a response, and similarly for pairs of responses. The fact that thread order is respected by linearizability follows from the general requirement that order from response to invocation must be respected.
– In addition to *returns*, we allow $s \in \mathsf{extensions}(s')$ to include *calls* that are not in $s'$. Assuming that specifications are prefix-closed, this permissiveness is harmless. For every spec $t$ that includes the extra calls in a suffix, there is a corresponding spec $t'$ such that $t \in \mathsf{extensions}(t')$ that does not include them; if $s'$ linearizes to $t$, then it also linearizes to $t'$.
– We require that all incomplete calls remain in $s'$. Assuming that specifications are input-enabled, this restriction is harmless. For every spec $t$ that does not include the extra calls, there is a corresponding spec $t' \in \mathsf{extensions}(t)$ that does include them; if $s'$ linearizes to $t$ with some incomplete calls removed, then it also linearizes to $t'$.

We can refactor the definition slightly to pull it into the shape used to define quiescent consistency and QQC.

*Definition 3.2.* For traces $s, t$, we write $s \sqsubseteq_{\mathsf{lin}} t$ if $s =_\pi t$ and for every prefix $p \leq_{\mathsf{pre}} s$

$$\forall a^! \in p.\ \forall b^? \in s - p.\ (a^! \Rightarrow_s b^?) \text{ implies } (a^! \Rightarrow_t b^?).$$

Then $(s'' \precsim_{\mathsf{lin}} t) \stackrel{\triangle}{=} (\exists s' \in \mathsf{extensions}(s'').\ \exists s =_\alpha s'.\ s \sqsubseteq_{\mathsf{lin}} t)$. □

*Lemma 3.3. s is linearizes to t iff $s \precsim_{\mathsf{lin}} t$.*

PROOF. This is an immediate consequence of the definition of prefix. □

This characterization of linearizability requires that we look at every way to *cut* the trace $s$ into a prefix $p$ and suffix $s - p$. We then look at the return events in $p$ and the call events in $s - p$ and ensure that the order of events *crossing the cut* is respected in $t$. The definitions are equivalent since we quantify over all possible cuts.

As an example, consider the incrementing counter specification from Example 1.1: $[^+\ ]^+_0\ \{^+\ \}^+_1\ (^+\ )^+_2$ . For a completely concurrent trace, such as $[^+\ \{^+\ (^+\ )^+_2\ \}^+_1\ ]^+_0$ linearizability is trivially satisfied since there is no cut that has a return on the left and call on the right. The trace $\{^+\ [^+\ \}^+_1\ (^+\ ]^+_0\ )^+_2$ is also linearizable. The interesting cut is $\{^+\ [^+\ \}^+_1$ which requires only that $\}^+_1$ precede $(^+$ in the specification. By the same reasoning, $\{^+\ (^+\ \}^+_1\ [^+\ )^+_2\ ]^+_0$ , is not linearizable, since it requires that $\}^+_1$ precede $[^+$ .

## 3.2 Second characterization: invocation to response

*Theorem 3.4. Let $t$ be a sequential trace with name order $(a^?_1, a^!_1, a^?_2, a^!_2, \ldots, a^?_n, a^!_n)$. Let $s$ be an operational trace such that $s =_\pi t$. Then*

$$s \sqsubseteq_{\mathsf{lin}} t \quad \textit{iff} \quad \forall j.\ \{1,\ldots,j\} \subseteq \{i \mid a^?_i \Rightarrow_s a^!_j\}$$

PROOF. Using the definition of linearizability and calculating, we have the following proof obligation.

$$(\forall i,j.\ a^!_i \Rightarrow_s a^?_j \text{ implies } i < j) \quad \Leftrightarrow \quad (\forall i,j.\ i \le j \text{ implies } a^?_i \Rightarrow_s a^!_j)$$

$(\Rightarrow)$ Fix $i \le j$. If $i = j$ the right implication holds by the definition of traces. Suppose $i < j$. By operationality, either $a^?_i \Rightarrow_s a^!_j$ or $a^!_j \Rightarrow_s a^?_i$. In the first case, the right implication holds. In the second case, the left implication requires $j < i$, a contradiction.

$(\Leftarrow)$ Fix $a^!_i \Rightarrow_s a^?_j$. By way of contradiction, suppose $i \le j$. From the right implication we deduce that $a^?_i \Rightarrow_s a^!_j$. The resulting cycle, $a^!_i \Rightarrow_s a^?_j \Rightarrow_s a^!_i$ contradicts the supposition that $s$ is a trace. Therefore it must be that $i < j$ as required. □

Let us revisit the incrementing counter specification $[^+\ ]^+_0\ \{^+\ \}^+_1\ (^+\ )^+_2$ . In the completely concurrent trace $[^+\ \{^+\ (^+\ )^+_2\ \}^+_1\ ]^+_0$ all invocations precede all responses, and therefore linearizability is trivially satisfied. The linearizability of $\{^+\ [^+\ \}^+_1\ (^+\ ]^+_0\ )^+_2$ follows from the fact that $\}^+_1$ is preceded by both $[^+$ and $\{^+$ , and the nonlinearizability of $\{^+\ (^+\ \}^+_1\ [^+\ )^+_2\ ]^+_0$ , follows from the fact that $[^+$ does not precede $\}^+_1$ .

## 3.3 Compositionality

We re-prove one of the fundamental properties of linearizability: compositionality [9]. The proof we give here is similar to the proof given for QQC in subsection 5.4, in a simpler setting.

*Lemma 3.5 (Operational traces). Suppose that $s$ is an operational trace that imposes the following order.*

$$\begin{tikzcd}
a^?_1 \arrow[dr] & a^?_0 \arrow[d] \arrow[drr] & & b^?_0 \arrow[d] \arrow[dll] & b^?_1 \arrow[dl] \\
& a^!_0 & & b^!_0 &
\end{tikzcd}$$

*Then either $a^?_1 \Rightarrow_s b^!_0$ or $b^?_1 \Rightarrow_s a^!_0$.*

PROOF. If neither holds, then, by operationality we must have both $b^{!}_0 \Rightarrow_s a^{?}_1$ and $a^{!}_0 \Rightarrow_s b^{?}_1$, which results in the cycle $b^{!}_0 \Rightarrow_s a^{?}_1 \Rightarrow_s a^{!}_0 \Rightarrow_s b^{?}_1 \Rightarrow_s b^{!}_0$. □

Recall from subsection 2.2 that $(|||)$ denotes interleaving and $(\div)$ denotes partial order difference. To split trace $s$ in "half," it suffices to postulate the existence of $s_1$ and $s_2$ such that $s_1 = s \div s_2$ and $s_2 = s \div s_1$.

*Theorem 3.6. Let $t_1$ and $t_2$ be sequential traces.*
*Let $s$, $s_1$ and $s_2$ be operational traces such that $s_1 = s \div s_2$ and $s_2 = s \div s_1$.*
*For $i \in \{1, 2\}$, suppose that each $s_i \sqsubseteq_{\mathsf{lin}} t_i$.*
*Then there exists a sequential trace $t \in (t_1 \mathbin{|||} t_2)$ such that $s \sqsubseteq_{\mathsf{lin}} t$.*

PROOF. Without loss of generality, assume that $\mathsf{ids}(t_1)$ and $\mathsf{ids}(t_2)$ are disjoint. Let the sequence of names in $t_1$ be $(a^{?}_1, a^{!}_1, \dots, a^{?}_m, a^{!}_m)$ and sequence of name in $t_2$ be $(b^{?}_1, b^{!}_1, \dots, b^{?}_n, b^{!}_n)$. Applying Theorem 3.4 to the supposition $s_1 \sqsubseteq_{\mathsf{lin}} t_1$, we have that $i \leq j$ implies $a^{?}_i \Rightarrow_s a^{!}_j$, and similarly for the $b$s.

Our aim is to construct a sequential interleaving of $t_1$ and $t_2$. To do this, we construct a partial order over event pairs. Any interleaving consistent with the partial order will satisfy the conclusion of the theorem by construction. For the elements of the partial order, let $a_i$ represent the pair $a^{?}_i a^{!}_i$ and let $b_k$ represent the pair $b^{?}_k b^{!}_k$. Let the $a$s be totally ordered by subscript, corresponding to the fact that $a^{?}_i \Rightarrow_s a^{!}_j$ whenever $i \leq j$, and similarly the $b$s. Let there be a *cross edge* from $a_i$ to $b_\ell$ if $a^{!}_i \Rightarrow_s b^{?}_\ell$, and symmetrically from $b$s to $a$s. Visually, we have an order such as the following.

$$\begin{tikzcd}
a_1 \arrow[r] & a_2 \arrow[r] & \cdots \arrow[r] & a_i \arrow[r] \arrow[drr] & \cdots \arrow[r] & a_j \arrow[r] & \cdots \arrow[r] & a_m \\
b_1 \arrow[r] & b_2 \arrow[r] \arrow[ul] & \cdots \arrow[r] & b_k \arrow[r] & \cdots \arrow[r] & b_\ell \arrow[r] & \cdots \arrow[r] & b_n
\end{tikzcd}$$

The $a$-$a$ and $b$-$b$ edges go from `?` to `!` in $s$, whereas the cross edges go from `!` to `?`.

The proof obligation is to show that this order is acyclic, in which case it induces at least one interleaving. We show that any cycle in the defined order corresponds to a cycle in $s$, contradicting the supposition that $s$ is a trace. For there to be a cycle in the defined order, there must be $i < j$ and $k < \ell$, such that $a^{?}_i \Rightarrow_s a^{!}_j \Rightarrow_s b^{?}_k \Rightarrow_s b^{!}_\ell \Rightarrow_s a^{?}_i$. This contradicts the supposition that $s$ is a trace. □

## 4 Quiescent Consistency

Let $\mathsf{open}(s)$ be the set of calls in $s$ that have no matching return[7]. We say that trace $s$ is *quiescent* if $\mathsf{open}(s) = \emptyset$. This notion of quiescence does not require that there be no active thread, but only that there be no open calls. Thus, this notion of quiescence is compatible with libraries that maintain their own thread pools.

The definition of quiescent consistency is similar to Definition 3.2 of linearizability. The difference lies in the quantifier for the prefix $p \leq_{\mathsf{pre}} s$: Whereas linearizability quantifies over *every* prefix, quiescent consistency only quantifies over *quiescent* prefixes.

[7] $\mathsf{open}(s) \triangleq \{u \in s \mid \mathsf{pol}(u) = \texttt{?} \wedge \nexists v \in s.\ \mathsf{brak}(v) = \mathsf{id}(u)\}$

*Definition 4.1.* We write $s \sqsubseteq_{\mathsf{qc}} t$ if $s =_{\pi} t$ and for any *quiescent* prefix $p \leq_{\mathsf{pre}} s$

$$\forall a^{!} \in p.\ \forall b^{?} \in s - p.\ (a^{!} \Rightarrow_{s} b^{?}) \text{ implies } (a^{!} \Rightarrow_{t} b^{?}).$$

Then $(s'' \underset{\sim}{\sqsubset}_{\mathsf{qc}} t) \triangleq (\exists s' \in \mathsf{extensions}(s'').\ \exists s =_{\alpha} s'.\ s \sqsubseteq_{\mathsf{qc}} t)$. □

Again let us revisit the counter specification from Example 1.1: $[^{+}\ ]^{+}_{0}\ \{^{+}\ \}^{+}_{1}\ (^{+}\ )^{+}_{2}$ . This notion quiescent consistency places some constraints on the system even when it has no nontrivial quiescent points. For example, the execution $[^{+}\ \{^{+}\ (^{+}\ )^{+}_{3}\ \}^{+}_{1}\ ]^{+}_{0}$ is not quiescently consistent with the given specification, since it is not a permutation. If one extends the execution to $[^{+}\ \{^{+}\ (^{+}\ )^{+}_{3}\ \}^{+}_{1}\ ]^{+}_{0}\ <^{+}\ >^{+}_{2}$ and attempts to matches it against the specification $[^{+}\ ]^{+}_{0}\ \{^{+}\ \}^{+}_{1}\ <^{+}\ >^{+}_{2}\ (^{+}\ )^{+}_{3}$ , quiescent consistency continues to fail: In the quiescent prefix $[^{+}\ \{^{+}\ (^{+}\ )^{+}_{3}\ \}^{+}_{1}\ ]^{+}_{0}$ , the order across the cut from $)^{+}_{3}$ to $<^{+}$ is not preserved in the specification.

For linearizability, we argued that because specifications are prefix-closed, only responses need be included in the extensions of a trace. The same does not hold for quiescent consistency. For example, since $(^{+}\ \{^{+}\ \}^{+}_{1}\ [^{+}\ ]^{+}_{0}\ )^{+}_{2}$ is quiescently consistent, its prefix $(^{+}\ \{^{+}\ \}^{+}_{1}$ should also be quiescently consistent. However, there is no specification trace that can be matched that does not include $[^{+}\ ]^{+}_{0}$ . Therefore, it does not suffice merely to close the open call by adding $)^{+}_{2}$ ; We must include $[^{+}$ and $]^{+}_{0}$ .

Compositionality (as expressed in Theorem 3.6) also holds for quiescent consistency. The proof is straightforward: any quiescent point of $s_1 \cup s_2$ is also a quiescent point for each $s_i$; the two specifications may be interleaved arbitrarily between these quiescent points.

As noted in the introduction, if the sequence of interlocking calls $[^{+}\ (^{+}\ ]^{+}_{i}\ [^{+}\ )^{+}_{j}\ (^{+}\ ]^{+}_{k}$ $[^{+}\ \cdots$ , never reaches quiescence, then the counter may return any natural number for $i$, $j$ and $k$. QQC reduces this permissiveness by looking at every cut. It remains less strict than linearizability by loosening the requirement that *every* response-to-invocation across the cut be respected in the specification.

## 5 Quantitative Quiescent Consistency

We provide three characterizations of QQC and prove their equivalence.

- In subsection 5.1, we define QQC in the style that we have defined linearizability and quiescent consistency, from response to invocation.
- In subsection 5.2, we give a *counting characterization* of QQC, which requires that if a response matches the $i^{th}$ method call in the specification, then it must be preceded by at least $i$ invocations.
- In subsection 5.3, we give a operational characterization of QQC as a proxy between the concurrent world and an underlying sequential data structure. This can be seen a mix of flat combining Hendler, Incze, Shavit, and Tzafrir [7] with speculation.

Finally, in subsection 5.4, we demonstrate that QQC is compositional, as in [9].

To give some intuition for the what is allowed by QQC, we first give some examples using the 2-`Counter` from the introduction. First we note that the capability given by an open call can be used repeatedly, as in the execution $(^{+}\ [^{+}\ ]^{+}_{1}\ \{^{+}\ \}^{+}_{0}\ [^{+}\ ]^{+}_{3}\ \{^{+}\ \}^{+}_{2}\ [^{+}\ ]^{+}_{5}\ \{^{+}\ \}^{+}_{4}\ )^{+}_{6}$ . Alternatively, multiple open calls may be accumulated to create an trace with events that are arbitrarily far off, as in $(^{+}\ [^{+}\ ]^{+}_{1}\ (^{+}\ [^{+}\ ]^{+}_{3}\ (^{+}\ [^{+}\ ]^{+}_{5}\ (^{+}\ [^{+}\ ]^{+}_{7}\ [^{+}\ ]^{+}_{0}\ )^{+}_{2}\ )^{+}_{4}\ )^{+}_{6}\ )^{+}_{8}$ . Note that $[^{+}\ ]^{+}_{0}$ *follows* $[^{+}\ ]^{+}_{7}$ in this execution! It is worth emphasizing that the order between these actions is observable to the outside: a single thread can call `getAndIncrement` and get 7, then subsequently call `getAndIncrement` and get 0.

In general, an $N$-`Counter` can give results that are $k \times N$ off of the expected value, where $k$ is the maximum number of open calls and $N$ is the width of the counter. There is no way to bound the behavior of this counter, as in [8], without also bounding the amount of concurrency, as in [1].

It is also possible for open calls to overlap in nontrivial ways. The trace $(^{+}\ [^{+}\ ]^{+}_{1}\ \{^{+}\ [^{+}\ ]^{+}_{0}\ )^{+}_{3}\ (^{+}\ )^{+}_{2}\ \}^{+}_{4}$ is QQC. Here, the first $(^{+}$ justifies the out-of-order execution of $[^{+}\ ]^{+}_{1}$ and $[^{+}\ ]^{+}_{0}$ . The subsequent $\{^{+}$ justifies an inversion of the previous justifier, namely $(^{+}\ )^{+}_{3}$ and $(^{+}\ )^{+}_{2}$ . A similar example is $\{^{+}\ (^{+}\ )^{+}_{1}\ (^{+}\ [^{+}\ ]^{+}_{0}\ )^{+}_{3}\ [^{+}\ ]^{+}_{2}\ \}^{+}_{4}$ .

Finally, we note that the stack execution $\{^{+}_{c}\ [^{-}\ ]^{-}_{a}\ (^{+}_{a}\ )^{+}\ \}^{+}$ is QQC with respect to the specification $(^{+}_{a}\ )^{+}\ [^{-}\ ]^{-}_{a}\ \{^{+}_{c}\ \}^{+}$ . This follows from exactly the kind of reasoning that we have done for the counter. This lack of causality may be troubling, but we note that it is typical of weak correctness criteria such as quiescent consistency. We revisit this issue in the conclusions.

## 5.1 First characterization: response to invocation

Linearizability requires that for *every* cut, *all* response-to-invocation order crossing the cut must be respected in the specification. Quiescent consistency limits this attention to *quiescent* cuts. QQC restores the quantification over every cut, but relaxes the requirement to match all response-to-invocation order crossing the cut. When checking response-to-invocation pairs across the cut, QQC allows some invocations to be ignored. How many?

One constraint comes from our desire to refine quiescent consistency. For quiescent cuts, we cannot drop any invocations, since this quiescent consistency does not. As a first attempt at a definition, we may take the number of dropped invocations at any cut to be bounded by $\left|\mathsf{open}(p)\right|$. However, this is too permissive. For example, this simple criterion cannot distinguish the following executions from Example 1.1.

$$(^{+}\ \{^{+}\ \}^{+}_{1}\ [^{+}\ ]^{+}_{0}\ )^{+}_{2} \qquad\qquad [^{+}\ (^{+}\ )^{+}_{2}\ \{^{+}\ \}^{+}_{1}\ ]^{+}_{0}$$

The interesting cut splits the traces in half at the midpoint. In each case there is one open call. Therefore in the first trace, we can ignore $[^{+}$ in the suffix, and in the second trace, we can ignore $\{^{+}$ in the suffix. However, we believe that these traces should be distinguished. The second trace is “more off” than the first.

The difference can be seen by looking not only at the number of open calls, but also at *which* calls are open. In the first trace we have $(^{+}$ before $\}^{+}_{1}$, and in the second, we have $[^{+}$ before $)^{+}_{2}$. We say that $(^{+}$ is *early* for $\}^{+}_{1}$, since it does not precede $\}^{+}_{1}$ in the specification, whereas $[^{+}$ is not early for $)^{+}_{2}$, since it *does* precede $)^{+}_{2}$. We restrict our attention to calls that are both open and early with respect to the response of interest.

Given a specification $t$ and $a^{!} \in t$ none of the actions in the $t$-downclosure of $a^{!}$ could possibly be early for $a^{!}$; any other action could be. Thus, the actions in $\mathsf{open}(p) - ({\downarrow_t}\, a^{!})$ are both open and early for $a^{!}$. This leads us to the following definition. (In subsection 5.2, we show that for sequential specifications, we can swap the quantifiers $(\exists r)$ and $(\forall a^{!})$, pulling out the existential.)

*Definition 5.1.* We write $s \sqsubseteq_{\mathsf{qqc}} t$ if $s =_{\pi} t$ and for any prefix $p \leq_{\mathsf{pre}} s$

$$\begin{aligned}&\forall a^{!} \in p.\ \exists r \subseteq s.\ \left|r\right| \leq \left|\mathsf{open}(p) - ({\downarrow_t}\, a^{!})\right|.\\ &\qquad \forall b^{?} \in ((s-p)-r).\ (a^{!} \Rightarrow_s b^{?}) \text{ implies } (a^{!} \Rightarrow_t b^{?}).\end{aligned}$$

Then $(s'' \underset{\sim}{\sqsubseteq}_{\mathsf{qqc}} t) \triangleq (\exists s' \in \mathsf{extensions}(s'').\ \exists s =_{\alpha} s'.\ s \sqsubseteq_{\mathsf{qqc}} t)$. □

In this definition, it is safe to restrict attention to sets $r$ consisting only of input events that are concurrent with the open calls. We do not impose these restrictions explicitly because they are not necessary. Choosing outputs does not add any flexibility, effectively wasting an open call. Non-concurrent calls will be revealed by the prefix in which the call is closed.

*Theorem 5.2.* $(\underset{\sim}{\sqsubseteq}_{\mathsf{lin}}) \subset (\underset{\sim}{\sqsubseteq}_{\mathsf{qqc}}) \subset (\underset{\sim}{\sqsubseteq}_{\mathsf{qc}})$

PROOF. Containment is immediate from the definitions, always taking $r = \varepsilon$ for QQC. To see that the containment is proper, consider the incrementing counter specification from Example 1.1, $[^{+}\ ]^{+}_{0}\ (^{+}\ )^{+}_{1}\ \{^{+}\ \}^{+}_{2}$ . With respect to this specification, $\{^{+}\ (^{+}\ )^{+}_{1}\ [^{+}\ ]^{+}_{0}\ \}^{+}_{2}$ is QQC but not linearizable $[^{+}\ \{^{+}\ \}^{+}_{2}\ (^{+}\ )^{+}_{1}\ ]^{+}_{0}$ is quiescently consistent but not QQC. □

## 5.2 Second characterization: counting invocations

Given the subtlety of Definition 5.1, it may be surprising that QQC has the following simple characterization for sequential specifications.

*Theorem 5.3. Let $t$ be a sequential trace with name order $(a^{?}_1, a^{!}_1, \ldots, a^{?}_n, a^{!}_n)$. Let $s$ be an operational trace such that $s =_{\pi} t$. Then*

$$s \sqsubseteq_{\mathsf{qqc}} t \quad \textit{iff} \quad \forall j.\ j \leq \left|\{a^{?}_i \mid a^{?}_i \Rightarrow_s a^{!}_j\}\right|$$

PROOF. $(\Rightarrow)$ Fix $j$, let $p = {\downarrow_s}\, a^{!}_j$, and let $q$, $r'$, $o$ be the following disjoint sets.

$$\begin{aligned}q &= \{a^{?}_i \mid i \leq j \wedge a^{?}_i \Rightarrow_s a^{!}_j\}\\ r' &= \{a^{?}_i \mid i \leq j \wedge a^{?}_i \not\Rightarrow_s a^{!}_j\} = \{a^{?}_i \mid i \leq j \wedge a^{!}_j \Rightarrow_s a^{?}_i\} && \text{(by operationality)}\\ o &= \{a^{?}_i \mid i > j \wedge a^{?}_i \Rightarrow_s a^{!}_j\} \supseteq \mathsf{open}(p) - ({\downarrow_t}\, a^{!}_j) && \text{(by calculation)}\end{aligned}$$

Note that $q \cup o = \{a^{?}_i \mid a^{?}_i \Rightarrow_s a^{!}_j\}$; therefore it suffices to show that $\left|q \cup o\right| \geq j$.

For every event in $a^{?}_i \in r'$ we have that $i \leq j$ and therefore $a^{!}_j \Rightarrow_s a^{?}_i$ and $a^{!}_j \not\Rightarrow_t a^{?}_i$. Hence the set $r$ chosen in Definition 5.1 must include $r'$. From Definition 5.1, we have that $\left|r\right| \leq \left|\mathsf{open}(p) - ({\downarrow_t}\, a^{!}_j) r\right|$. Since $r' \subseteq r$ and $\mathsf{open}(p) - ({\downarrow_t}\, a^{!}_j) \subseteq o$, we have $\left|r'\right| \leq \left|o\right|$. Since $\left|q \cup r'\right| = j$, we have $\left|q \cup o\right| \geq j$, as required.

($\Leftarrow$) Fix $p$. Following the argument given in the proof of Lemma 5.4, in order to show that the requirements of Definition 5.1 hold for every $a^! \in p$, it suffices to show that they hold for $a^!_j$, where let $j = \max\{k \mid a^!_k \in p\}$.

Fix $j = \max\{k \mid a^!_k \in p\}$. We now show that the requirements of Definition 5.1 hold for $a^!_j$. We choose $q$, $r$ and $o$ as before.

$$\begin{aligned}
q &= \{a^?_i \mid i \leq j \wedge a^?_i \Rightarrow_s a^!_j\} \\
r &= \{a^?_i \mid i \leq j \wedge a^?_i \not\Rightarrow_s a^!_j\} = \{a^?_i \mid i \leq j \wedge a^!_j \Rightarrow_s a^?_i\} \\
o &= \{a^?_i \mid i > j \wedge a^?_i \Rightarrow_s a^!_j\} \subseteq \mathsf{open}(p) - (\downarrow_t a^!_j)
\end{aligned}$$

To see that $o \subseteq \mathsf{open}(p)$, consider that if $a^?_i \in o$ then $a^!_i \notin p$; otherwise $j \neq \max\{k \mid a^!_k \in p\}$. By the second characterization of $r$ above (which follows from operationality), $\forall a^?_i \notin r.\ (a^!_j \Rightarrow_s a^?_i)$ implies $j < i$. Thus, to establish the result it suffices to show that $|r| \leq |\mathsf{open}(p) - (\downarrow_t a^!_j)|$. By assumption, $|q \cup o| \geq j$. Since $|q \cup r| = j$, we have $|r| \leq |o|$ and therefore $|r| \leq |\mathsf{open}(p) - (\downarrow_t a^!_j)|$ as required. $\square$

This characterization provides a simple method for calculating whether a trace is QQC. For example the trace $\{^+\ (^+\ )^+_1\ (^+\ [^+\ ]^+_0\ )^+_3\ [^+\ ]^+_2\ \}^+_4$ is QQC since $)^+_1$ is preceded by two calls, $]^+_0$ , $)^+_3$ by four, and $]^+_2$ , $\}^+_4$ by five. The trace $\{^+\ (^+\ )^+_1\ (^+\ )^+_3\ [^+\ ]^+_0\ [^+\ ]^+_2\ \}^+_4$ is not QQC since $)^+_3$ is only preceded by three calls, yet it is the fourth call in the specification.

For sequential specifications, we can also simplify Definition 5.1 by exchanging the quantifiers $(\exists r)$ and $(\forall a^!)$, pulling out the existential.

*Lemma 5.4. Let $t$ be a sequential trace with name order $(a^?_1, a^!_1, \ldots, a^?_n, a^!_n)$. Let $s$ be an operational trace such that $s =_\pi t$. Fix $p \leq_{\mathsf{pre}} s$. Then the displayed requirement of Definition 5.1 is equivalent to*

$$\begin{aligned}
&\exists r \subseteq s.\ |r| \leq |\mathsf{openEarly}_t(p)|. \\
&\qquad \forall a^! \in p.\ \forall b^? \in ((s - p) - r).\ (a^! \Rightarrow_s b^?) \textit{ implies } (a^! \Rightarrow_t b^?),
\end{aligned}$$

*where* $\mathsf{openEarly}_t(p) \triangleq \{b^? \in \mathsf{open}(p) \mid \nexists a^! \in p.\ b^? \Rightarrow_t a^!\}$.

PROOF. ($5.4 \Rightarrow 5.1$) Immediate.

($5.1 \Rightarrow 5.4$) Consider the proof of the reverse direction ($\Leftarrow$) in the Theorem 5.3. An examination of the proof shows that the open calls constructed satisfy the more stringent requirements of 5.4. In fact, the proof of 5.3 shows that ($5.3 \Rightarrow 5.4$). The result follows since the forward direction of 5.3 shows that ($5.1 \Rightarrow 5.3$). $\square$

For full concurrent specifications and implementations, we suspect that Lemma 5.4 fails. (To get a sense of the issues, consider a specification that orders $a \longrightarrow c$ and $b \longrightarrow d$, and an implementation that executes $a \longrightarrow d$ and $b \longrightarrow c$.) In this paper, however, all of our results concern sequential specifications and operational implementations.

## 5.3 Third characterization: speculative flat combining

Our third characterization of QQC describes how QQC affects an arbitrary sequential data structure, using a *proxy* that generates QQC traces from an underlying sequential implementation. The proxy is *sound*, in that every trace that it accepts is QQC, and *complete*, in that it generates every operational trace that is QQC with respect to the sequential data structure.

This characterization of QQC incorporates *speculation* into flat combining [7]. *Flat combining* is a technique for implementing concurrent data structures using sequential ones by introducing a mediator between the concurrent world and the sequential data structure. As for speculation, we push the obligation to predict the future into the underlying sequential object, with must conform to the following interface.

```
interface Object {
   method run(i:Invocation):Response;
   method predict():Invocation;  }
```

The `run` method passes invocations to the underlying sequential structure and returns the appropriate response. The `predict` method is an oracle that guesses the invocations that are to come in the future. It is the use of `predict` that makes our code speculative.

Given an `Object` `o`, the proxy is defined as follows.

```
class QQCProxy<o:Object> {
   field called:ThreadSafeMultiMap<Invocation,Semaphore> = [];
   field returned:ThreadSafeMap   <Semaphore, Response>  = [];
   method run(i:Invocation):Response { // proxy for external access to o
     val m:Semaphore = [];
     called.add(i, m);
     m.wait();
     return returned.remove(m); }
   thread { // single thread to interact with o
     val received:MultiMap<Invocation,Semaphore> = [];
     val executed:MultiMap<Invocation,Response>  = [];
     repeatedly choose {
        choice if called.notEmpty() {
           received.add(called.removeAny());
           val i:Invocation = o.predict();
           val r:Response   = o.run(i);
           executed.add(i, r); }
        choice if exists i in received.keys() intersect executed.keys() {
           val m:Semaphore = received.remove(i);
           val r:Response  = executed.remove(i);
           returned.add(m, r);
           m.signal(); } } } }
```

Communication between the implementation threads and the underlying `Object` is mediated by two maps. When a thread would like to interact with the `Object`, it creates a semaphore, registers the semaphore in `called` and waits on the semaphore. Upon awakening, the thread removes the result from `returned` and returns.

The `Object` is serviced by a single *proxy* thread which loops forever making one of two nondeterministic choices. The proxy keeps two private maps. Upon receiving an called invocation, the proxy moves the invocation from `called` to `received`. Rather than executing the received invocation, the proxy asks the oracle to predict an arbitrary invocation `i` and executes that instead, placing the result in `executed`. Once a invocation is both `received` and `executed`, it may become `returned`.

At the beginning of this section, we noted that the stack execution $\{^{+}_{c}\ [^{-}\ ]^{-}_{a}\ (^{+}_{a}\ )^{+}\ \}^{+}$ is QQC with respect to the specification $(^{+}_{a}\ )^{+}\ [^{-}\ ]^{-}_{a}\ \{^{+}_{c}\ \}^{+}$. How can such a trace possibly be generated? The execution of the proxy proceeds as follows. Upon receipt of $\{^{+}_{c}$, the proxy executes $(^{+}_{a}$, storing the response $)^{+}$. Upon receipt of $[^{-}$, the proxy executes $[^{-}$, storing the response $]^{-}_{a}$. At this point $[^{-}\ ]^{-}_{a}$ can be returned. Upon receipt of $(^{+}_{a}$, the proxy executes $\{^{+}_{c}$, storing the response $\}^{+}$. At this point both $(^{+}_{a}\ )^{+}$ and $\{^{+}_{c}\ \}^{+}$ can be returned.

*Theorem 5.5. The concurrent proxy is sound for QQC with respect to the underlying `Object`. It is also complete for operational traces.*

PROOF. For soundness, note that proxy maintains the invariant that the sizes of `received` and `executed` are equal, and therefore the number of returned calls can never exceed the number that has been received. In addition, the number of things added to `received` always exceeds the number added to `returned`.

For completeness, suppose that trace $s \sqsubseteq_{\mathsf{qqc}} t$ and let the sequence of names in $t$ be $(a^{?}_{1}, a^{!}_{1}, \ldots, a^{?}_{m}, a^{!}_{m})$. Consider any total order on the events of $s$ that is consistent with the order already present in $t$. Let $(b^{?}_{1}, \ldots, b^{?}_{m})$ be the order on the call actions in this total order. When $b^{?}_{i}$ arrives, add $b^{?}_{i}$ to `received` and execute $a^{?}_{i}$, placing $a^{!}_{i}$ into `executed`. From Theorem 5.3 we know that whenever a response is required, there will be enough prior invocations so that the required response will be found in `executed`.□

## 5.4 Compositionality

We now prove compositionality for QQC, following the proof for linearizability in Theorem 3.6. Below, we give some examples of the construction given in the proof, which is more complex than the one required for linearizability. Recall that $(\div)$ denotes partial order difference.

*Theorem 5.6. Let $t_1$ and $t_2$ be sequential traces.*

*Let $s$, $s_1$ and $s_2$ be operational traces such that $s_1 = s \div s_2$ and $s_2 = s \div s_1$.*

*For $i \in \{1, 2\}$, suppose that each $s_i \sqsubseteq_{\mathsf{qqc}} t_i$.*

*Then there exists a sequential trace $t \in (t_1 \mathbin{|||} t_2)$ such that $s \sqsubseteq_{\mathsf{qqc}} t$.*

PROOF. As in the proof of Theorem 3.6, assume $\mathsf{ids}(t_1)$ and $\mathsf{ids}(t_2)$ are disjoint, and let the sequence of names in $t_1$ be $(a^{?}_{1}, a^{!}_{1}, \ldots, a^{?}_{m}, a^{!}_{m})$ and sequence of name in $t_2$ be $(b^{?}_{1}, b^{!}_{1}, \ldots, b^{?}_{n}, b^{!}_{n})$. Applying Theorem 5.3 to the supposition $s_1 \sqsubseteq_{\mathsf{lin}} t_1$, we have that $j \leq \left|\{a^{?}_{i} \mid a^{?}_{i} \Rightarrow_{s} a^{!}_{j}\}\right|$, and similarly $\ell \leq \left|\{b^{?}_{k} \mid b^{?}_{k} \Rightarrow_{s} b^{!}_{\ell}\}\right|$. It suffices to construct an interleaving $t \in (t_1 \mathbin{|||} t_2)$ such that whenever $t$ contains a subsequence with names

$$a^{?}_{j}, a^{!}_{j}, b^{?}_{k}, b^{!}_{k}, b^{?}_{k+1}, b^{!}_{k+1}, \ldots, b^{?}_{k+x}, b^{!}_{k+x}$$

then for every $k \leq \ell \leq k+x$, we have

$$\{a^{?}_{i} \mid a^{?}_{i} \Rightarrow_{s} a^{!}_{j}\} \subseteq \{a^{?}_{i} \mid a^{?}_{i} \Rightarrow_{s} b^{!}_{\ell}\}$$

and symmetrically for subsequences $b^{?}_{k}, b^{!}_{k}, a^{?}_{j}, a^{!}_{j}, a^{?}_{j+1}, a^{!}_{j+1}, \ldots, a^{?}_{j+y}, a^{!}_{j+y}$. Given such a $t$, we know that $j + \ell \leq \left|\{a^{?}_{i} \mid a^{?}_{i} \Rightarrow_{s} b^{!}_{\ell}\} \cup \{b^{?}_{k} \mid b^{?}_{k} \Rightarrow_{s} b^{!}_{\ell}\}\right|$, as required.

We now demonstrate the existence of such a $t$. Define the set $\mathsf{merge}(\vec{a}, \vec{b})$ as follows.

$$\begin{array}{ll}
\mathsf{merge}(\vec{a},\, \varepsilon) = \{\vec{a}\} & \mathsf{merge}(\,\varepsilon\,, \vec{b}) = \{\vec{b}\} \\
\mathsf{merge}(\vec{a}\, a^?_j\, a^!_j,\ \vec{b}\, b^?_\ell\, b^!_\ell) \ni \vec{c}\, b^?_\ell\, b^!_\ell & \text{if } \vec{c} \in \mathsf{merge}(\vec{a}\, a^?_j\, a^!_j,\ \vec{b}) \\
 & \text{and } \{a^?_i \mid a^?_i \Rightarrow_s a^!_j\} \subseteq \{a^?_i \mid a^?_i \Rightarrow_s b^!_\ell\} \\
\mathsf{merge}(\vec{a}\, a^?_j\, a^!_j,\ \vec{b}\, b^?_\ell\, b^!_\ell) \ni \vec{c}\, a^?_j\, a^!_j & \text{if } \vec{c} \in \mathsf{merge}(\vec{a},\ \vec{b}\, b^?_\ell\, b^!_\ell) \\
 & \text{and } \{b^?_k \mid b^?_k \Rightarrow_s b^!_\ell\} \subseteq \{b^?_k \mid b^?_k \Rightarrow_s a^!_j\}
\end{array}$$

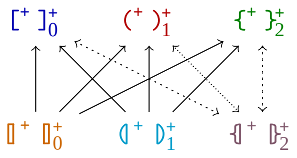
To demonstrate the existence of an appropriate $t$, it suffices to show that $\mathsf{merge}(a^?_1\, a^!_1 \ldots a^?_m\, a^!_m,\ b^?_1\, b^!_1 \ldots b^?_n\, b^!_n)$ is nonempty. By operationality, it must be the case that either (1) $a^!_j \Rightarrow_s b^!_\ell$, in which case $\{a^?_i \mid a^?_i \Rightarrow_s a^!_j\} \subseteq \{a^?_i \mid a^?_i \Rightarrow_s b^!_\ell\}$, (2) $b^!_\ell \Rightarrow_s a^!_j$, in which case $\{b^?_k \mid b^?_k \Rightarrow_s b^!_\ell\} \subseteq \{b^?_k \mid b^?_k \Rightarrow_s a^!_j\}$, or (3) $a^!_j$ and $b^!_\ell$ are unordered, in which case both conclusions hold. Therefore an appropriate $t$ exists. $\square$

*Example 5.7.* We demonstrate the $\mathsf{merge}$ function defined in the proof above using the following traces.

$$\begin{array}{ll}
t_1 = [^+\ ]^+_0\ (^+\ )^+_1\ \{^+\ \}^+_2 & t_2 = \llbracket^+\ \rrbracket^+_0\ \llparenthesis^+\ \rrparenthesis^+_1\ \lBrace^+\ \rBrace^+_2 \\
s_1 = \{^+\ (^+\ )^+_1\ [^+\ ]^+_0\ \}^+_2 & s_2 = \lBrace^+\ \llparenthesis^+\ \rrparenthesis^+_1\ \llbracket^+\ \rrbracket^+_0\ \rBrace^+_2
\end{array}$$

$$s = \lBrace^+\ \llparenthesis^+\ \rrparenthesis^+_1\ \{^+\ \llbracket^+\ \rrbracket^+_0\ (^+\ )^+_1\ [^+\ ]^+_0\ \}^+_2\ \rBrace^+_2$$

In the graph below, we draw an edge from $a_j$ to $b_\ell$ if $\{a^?_i \mid a^?_i \Rightarrow_s a^!_j\} \subseteq \{a^?_i \mid a^?_i \Rightarrow_s b^!_\ell\}$, indicating that $b_\ell$ may come after $a_j$. Edges from $b_\ell$ to $a_j$ are similar. When an edge is bidirectional, we use a dashed line.


$[^+\ ]^+_0$
$(^+\ )^+_1$
$\{^+\ \}^+_2$
$\llbracket^+\ \rrbracket^+_0$
$\llparenthesis^+\ \rrparenthesis^+_1$
$\lBrace^+\ \rBrace^+_2$

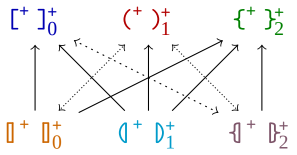
The following traces are derived from the $\mathsf{merge}$ algorithm.

$$\begin{array}{c}
\llbracket^+\ \rrbracket^+_0\ \llparenthesis^+\ \rrparenthesis^+_1\ \lBrace^+\ \rBrace^+_2\ [^+\ ]^+_0\ (^+\ )^+_1\ \{^+\ \}^+_2 \\
\llbracket^+\ \rrbracket^+_0\ \llparenthesis^+\ \rrparenthesis^+_1\ [^+\ ]^+_0\ \lBrace^+\ \rBrace^+_2\ (^+\ )^+_1\ \{^+\ \}^+_2 \\
\llbracket^+\ \rrbracket^+_0\ \llparenthesis^+\ \rrparenthesis^+_1\ [^+\ ]^+_0\ (^+\ )^+_1\ \lBrace^+\ \rBrace^+_2\ \{^+\ \}^+_2 \\
\llbracket^+\ \rrbracket^+_0\ \llparenthesis^+\ \rrparenthesis^+_1\ [^+\ ]^+_0\ (^+\ )^+_1\ \{^+\ \}^+_2\ \lBrace^+\ \rBrace^+_2
\end{array}$$

Suppose instead that we have the following $s$.

$$s = \lBrace^+\ \llparenthesis^+\ \rrparenthesis^+_1\ \{^+\ \llbracket^+\ (^+\ )^+_1\ \rrbracket^+_0\ \rBrace^+_2\ [^+\ ]^+_0\ \}^+_2$$

Then the graph and resulting traces are as follows.


$[^+\ ]^+_0$
$(^+\ )^+_1$
$\{^+\ \}^+_2$
$\llbracket^+\ \rrbracket^+_0$
$\llparenthesis^+\ \rrparenthesis^+_1$
$\lBrace^+\ \rBrace^+_2$

$$\begin{array}{c}
\llbracket^+\ \rrbracket^+_0\ \llparenthesis^+\ \rrparenthesis^+_1\ \lBrace^+\ \rBrace^+_2\ [^+\ ]^+_0\ (^+\ )^+_1\ \{^+\ \}^+_2 \\
\llbracket^+\ \rrbracket^+_0\ \llparenthesis^+\ \rrparenthesis^+_1\ [^+\ ]^+_0\ \lBrace^+\ \rBrace^+_2\ (^+\ )^+_1\ \{^+\ \}^+_2 \\
\llbracket^+\ \rrbracket^+_0\ \llparenthesis^+\ \rrparenthesis^+_1\ [^+\ ]^+_0\ (^+\ )^+_1\ \lBrace^+\ \rBrace^+_2\ \{^+\ \}^+_2
\end{array}$$

In general, if one where to include the linear order from the specification (eg, from $[^{+}\,]^{+}_{0}$ to $(^{+}\,)^{+}_{1}$), the resulting graph might be cyclic, even if the dotted edges were removed.□

## 6 Stack example

We show that, under reasonable assumptions, our $N$-`Stack` is QQC. We extend this argument to the elimination-tree stacks of [13].

In proving that executions of our $N$-`Stack` are QQC, the key step is to generate the corresponding specification trace. To do so, we consider the following instrumentation.

```
1  class Stack<N:Int> {
2    field b:[0..N-1] = 0;                     // 1 balancer
3    field s:Stack[]  = [[], [], ..., []]; // N stacks of values
4    field e:[0..N-1] = 0;                     // 1 emitter
5    field q:Queue[]  = [[], [], ..., []]; // N queues of actions
6    method push(x:Object):Unit {
7      val i:[0..N-1];
8      atomic {i=b; b++;}
9      atomic {val v=s[i].push(x); q[i].add("push" x); emit(); return v;} }
10   method pop():Object {
11     val i:[0..N-1];
12     atomic {i=b-1; b--;}
13     atomic {val v=s[i].pop(); q[i].add("pop" v); emit(); return v;} }
14   method emit():Unit {
15     while (q[e].first()=~"push" || q[e-1].first()=~"pop") {
16       if (q[e].first()=~"push")  {print (q[e].remove());   e++;}
17       if (q[e-1].first()=~"pop") {print (q[e-1].remove()); e--;} } } }
```

The state of the machine includes the values of the balancer `b` and stacks `s`. It also includes queues `q` to store the actions that have been executed on each stack and a *emitter* `e`, with the same range as `b`, which indicates the queue that should produce the next specification action. The emitter prints any completed pushes from `s[e]` and any completed pops from `s[e-1]`. When the emitter prints a push, it removes it from the queue and increments `e`; when it prints a pop, it removes it from the queue and decrements `e`. Emitter actions take place as soon as possible, and the emitter continues until it has nothing left to do.

Atomic blocks can only execute concurrently if they do not touch the same shared state. For the code in the introduction, this imposes an order between all executions of the first atomic (lines 8 and 12), since they touch the shared variable `b`; order is only imposed between executions of the second atomic that update the same stack. The presence of `emit` indicates also imposes an order between all executions of the second atomic (lines 9 and 13), since `emit` touches the shared variable `e`. This total order on calls to `emit` ensures that the printed trace is indeed a stack trace, as we argue below.

*Definition 6.1.* Let $a$ be a call to `push` or `pop`. Then $\mathsf{time}_1(a)$ is the time of the execution of the first atomic statement in the $N$-`Stack`, and $\mathsf{time}_2(a)$ is the time of the execution of the second atomic. A *linearized trace* of an $N$-`Stack` is one in which the invocations are ordered consistently with $\mathsf{time}_1$ and the responses are ordered consistently with $\mathsf{time}_2$. □

For example, from the linearization $(^{+}_{b}\ [^{+}_{a}\ ]^{+}\ )^{+}$ we know $\mathsf{time}_1((^{+}_{b}) < \mathsf{time}_1([^{+}_{a})$ and $\mathsf{time}_2(]^{+}) < \mathsf{time}_2()^{+})$. Such a linearized trace is distinct from other linearizations of the same trace, such as $(^{+}_{b}\ [^{+}_{a}\ )^{+}\ ]^{+}$, $[^{+}_{a}\ (^{+}_{b}\ ]^{+}\ )^{+}$ and $[^{+}_{a}\ (^{+}_{b}\ )^{+}\ ]^{+}$.

The response order in the linearized trace is particularly significant. For example, the linearization $(^{+}_{b}\ [^{+}_{a}\ ]^{+}\ )^{+}\ [^{-}\ ]^{-}_{a}\ (^{-}\ )^{-}_{b}$ cannot result from the execution of a 1-`Stack`. In this case $a$ is pushed before $b$ and therefore the pop of $a$ cannot be ordered before the pop of $b$.

*Example 6.2.* Consider the following linearized trace of a 2-`Stack`.

$$(^{+}_{c}\ <^{+}_{b}\ >^{+}\ [^{+}_{a}\ ]^{+}\ )^{+}\ (^{-}\ )^{-}_{c}\ <^{-}\ >^{-}_{b}\ [^{-}\ ]^{-}_{a}$$

Execution proceeds as follows. We show the atomic that is being executed above the arrow. Arrows without labels are executed within `emit`, atomically with the prior label. On the right-hand side, we show any emitted actions, followed by the resulting state. The initial state of the machine is $\langle \mathsf{b}=0, \mathsf{e}=0, \mathsf{s}=[[\,],[\,]], \mathsf{q}=[[], []]\rangle$.

$$\begin{array}{rll}
 & & \langle \mathsf{b}=0, \mathsf{e}=0, \mathsf{s}=[[\,],[\,]], & \mathsf{q}=[[], []]\rangle \\
\xrightarrow{(^{+}_{c}} & & \langle \mathsf{b}=1, \mathsf{e}=0, \mathsf{s}=[[\,],[\,]], & \mathsf{q}=[[], []]\rangle \\
\xrightarrow{<^{+}_{b}} & & \langle \mathsf{b}=0, \mathsf{e}=0, \mathsf{s}=[[\,],[\,]], & \mathsf{q}=[[], []]\rangle \\
\xrightarrow{>^{+}} & & \langle \mathsf{b}=0, \mathsf{e}=0, \mathsf{s}=[[\,],[b]], & \mathsf{q}=[[], [<^{+}_{b}\ >^{+}]]\rangle \\
\xrightarrow{[^{+}_{a}} & & \langle \mathsf{b}=1, \mathsf{e}=0, \mathsf{s}=[[\,],[b]], & \mathsf{q}=[[], [<^{+}_{b}\ >^{+}]]\rangle \\
\xrightarrow{]^{+}} & & \langle \mathsf{b}=1, \mathsf{e}=0, \mathsf{s}=[[a],[b]], & \mathsf{q}=[[[^{+}_{a}\ ]^{+}], [<^{+}_{b}\ >^{+}]]\rangle \\
\longrightarrow & [^{+}_{a}\ ]^{+} & \langle \mathsf{b}=1, \mathsf{e}=1, \mathsf{s}=[[a],[b]], & \mathsf{q}=[[], [<^{+}_{b}\ >^{+}]]\rangle \\
\longrightarrow & <^{+}_{b}\ >^{+} & \langle \mathsf{b}=1, \mathsf{e}=0, \mathsf{s}=[[a],[b]], & \mathsf{q}=[[], []]\rangle \\
\xrightarrow{)^{+}} & & \langle \mathsf{b}=1, \mathsf{e}=0, \mathsf{s}=[[ca],[b]], & \mathsf{q}=[[(^{+}_{c}\ )^{+}], []]\rangle \\
\longrightarrow & (^{+}_{c}\ )^{+} & \langle \mathsf{b}=1, \mathsf{e}=1, \mathsf{s}=[[ca],[b]], & \mathsf{q}=[[], []]\rangle \\
\xrightarrow{(^{-}} & & \langle \mathsf{b}=0, \mathsf{e}=1, \mathsf{s}=[[ca],[b]], & \mathsf{q}=[[], []]\rangle \\
\xrightarrow{)^{-}_{c}} & & \langle \mathsf{b}=0, \mathsf{e}=1, \mathsf{s}=[[a],[b]], & \mathsf{q}=[[(^{-}\ )^{-}_{c}], []]\rangle \\
\longrightarrow & (^{-}\ )^{-}_{c} & \langle \mathsf{b}=0, \mathsf{e}=0, \mathsf{s}=[[a],[b]], & \mathsf{q}=[[], []]\rangle \\
\xrightarrow{<^{-}} & & \langle \mathsf{b}=1, \mathsf{e}=0, \mathsf{s}=[[a],[b]], & \mathsf{q}=[[], []]\rangle \\
\xrightarrow{>^{-}_{b}} & & \langle \mathsf{b}=1, \mathsf{e}=0, \mathsf{s}=[[a],[\,]], & \mathsf{q}=[[], [<^{-}\ >^{-}_{b}]]\rangle \\
\longrightarrow & <^{-}\ >^{-}_{b} & \langle \mathsf{b}=1, \mathsf{e}=1, \mathsf{s}=[[a],[\,]], & \mathsf{q}=[[], []]\rangle \\
\xrightarrow{[^{-}} & & \langle \mathsf{b}=0, \mathsf{e}=1, \mathsf{s}=[[a],[\,]], & \mathsf{q}=[[], []]\rangle \\
\xrightarrow{]^{-}_{a}} & & \langle \mathsf{b}=0, \mathsf{e}=1, \mathsf{s}=[[\,],[\,]], & \mathsf{q}=[[[^{-}\ ]^{-}_{a}], []]\rangle \\
\longrightarrow & [^{-}\ ]^{-}_{a} & \langle \mathsf{b}=0, \mathsf{e}=0, \mathsf{s}=[[\,],[\,]], & \mathsf{q}=[[], []]\rangle
\end{array}$$

□

*Example 6.3.* Consider the following execution of the instrumented counter.

$$\begin{array}{rll}
 & \langle \mathsf{b}=0, \mathsf{e}=0, \mathsf{s}=[[\,],[\,]], & \mathsf{q}=[[], []]\rangle \\
\xrightarrow{[^{+}_{0}} & \langle \mathsf{b}=1, \mathsf{e}=0, \mathsf{s}=[[\,],[\,]], & \mathsf{q}=[[], []]\rangle \\
\xrightarrow{(^{+}_{a}} & \langle \mathsf{b}=0, \mathsf{e}=0, \mathsf{s}=[[\,],[\,]], & \mathsf{q}=[[], [(^{+}_{a}\ )^{+}]]\rangle
\end{array}$$

$$\begin{array}{lll}
\xrightarrow{)^{+}} & & \langle \mathtt{b}=0, \mathtt{e}=0, \mathtt{s}=[[\,],[a]], \quad \mathtt{q}=[[],[{(}^{+}_{a}\,{)}^{+}]]\rangle \\
\xrightarrow{(^{-}} & & \langle \mathtt{b}=1, \mathtt{e}=0, \mathtt{s}=[[\,],[a]], \quad \mathtt{q}=[[],[{(}^{+}_{a}\,{)}^{+}]]\rangle \\
\xrightarrow{)^{-}_{a}} & & \langle \mathtt{b}=1, \mathtt{e}=0, \mathtt{s}=[[\,],[\,]], \quad \mathtt{q}=[[],[{(}^{+}_{a}\,{)}^{+}\,{(}^{-}\,{)}^{-}_{a}]]\rangle \\
\xrightarrow{(^{+}_{b}} & & \langle \mathtt{b}=0, \mathtt{e}=0, \mathtt{s}=[[\,],[\,]], \quad \mathtt{q}=[[],[{(}^{+}_{a}\,{)}^{+}\,{(}^{-}\,{)}^{-}_{a}]]\rangle \\
\xrightarrow{)^{+}} & & \langle \mathtt{b}=0, \mathtt{e}=0, \mathtt{s}=[[\,],[b]], \quad \mathtt{q}=[[],[{(}^{+}_{a}\,{)}^{+}\,{(}^{-}\,{)}^{-}_{a}\,{(}^{+}_{b}\,{)}^{+}]]\rangle \\
\xrightarrow{[^{+}_{2}} & & \langle \mathtt{b}=1, \mathtt{e}=0, \mathtt{s}=[[\,],[b]], \quad \mathtt{q}=[[],[{(}^{+}_{a}\,{)}^{+}\,{(}^{-}\,{)}^{-}_{a}\,{(}^{+}_{b}\,{)}^{+}]]\rangle \\
\xrightarrow{(^{+}_{c}} & & \langle \mathtt{b}=0, \mathtt{e}=0, \mathtt{s}=[[\,],[b]], \quad \mathtt{q}=[[],[{(}^{+}_{a}\,{)}^{+}\,{(}^{-}\,{)}^{-}_{a}\,{(}^{+}_{b}\,{)}^{+}]]\rangle \\
\xrightarrow{)^{+}} & & \langle \mathtt{b}=0, \mathtt{e}=0, \mathtt{s}=[[\,],[bc]], \quad \mathtt{q}=[[],[{(}^{+}_{a}\,{)}^{+}\,{(}^{-}\,{)}^{-}_{a}\,{(}^{+}_{b}\,{)}^{+}\,{(}^{+}_{c}\,{)}^{+}]]\rangle \\
\xrightarrow{]^{+}} & & \langle \mathtt{b}=0, \mathtt{e}=0, \mathtt{s}=[[0],[bc]], \quad \mathtt{q}=[[{[}^{+}_{0}\,{]}^{+}],[{(}^{+}_{a}\,{)}^{+}\,{(}^{-}\,{)}^{-}_{a}\,{(}^{+}_{b}\,{)}^{+}\,{(}^{+}_{c}\,{)}^{+}]]\rangle \\
\longrightarrow & {[}^{+}_{0}\,{]}^{+} & \langle \mathtt{b}=0, \mathtt{e}=1, \mathtt{s}=[[0],[bc]], \quad \mathtt{q}=[[],[{(}^{+}_{a}\,{)}^{+}\,{(}^{-}\,{)}^{-}_{a}\,{(}^{+}_{b}\,{)}^{+}\,{(}^{+}_{c}\,{)}^{+}]]\rangle \\
\longrightarrow & {(}^{+}_{a}\,{)}^{+} & \langle \mathtt{b}=0, \mathtt{e}=0, \mathtt{s}=[[0],[bc]], \quad \mathtt{q}=[[],[{(}^{-}\,{)}^{-}_{a}\,{(}^{+}_{b}\,{)}^{+}\,{(}^{+}_{c}\,{)}^{+}]]\rangle \\
\longrightarrow & {(}^{-}\,{)}^{-}_{a} & \langle \mathtt{b}=0, \mathtt{e}=1, \mathtt{s}=[[0],[bc]], \quad \mathtt{q}=[[],[{(}^{+}_{b}\,{)}^{+}\,{(}^{+}_{c}\,{)}^{+}]]\rangle \\
\longrightarrow & {(}^{+}_{b}\,{)}^{+} & \langle \mathtt{b}=0, \mathtt{e}=0, \mathtt{s}=[[0],[bc]], \quad \mathtt{q}=[[],[{(}^{+}_{c}\,{)}^{+}]]\rangle \\
\xrightarrow{]^{+}} & & \langle \mathtt{b}=0, \mathtt{e}=0, \mathtt{s}=[[01],[bc]], \quad \mathtt{q}=[[{[}^{+}_{1}\,{]}^{+}],[{(}^{+}_{c}\,{)}^{+}]]\rangle \\
\longrightarrow & {[}^{+}_{1}\,{]}^{+} & \langle \mathtt{b}=0, \mathtt{e}=1, \mathtt{s}=[[01],[bc]], \quad \mathtt{q}=[[],[{(}^{+}_{c}\,{)}^{+}]]\rangle \\
\longrightarrow & {(}^{+}_{c}\,{)}^{+} & \langle \mathtt{b}=0, \mathtt{e}=0, \mathtt{s}=[[01],[bc]], \quad \mathtt{q}=[[],[]]\rangle
\end{array}$$

This produces the following linearized trace $s$ and specification $t$.

$$\begin{aligned}
s &= {[}^{+}_{0}\,{(}^{+}_{a}\,{)}^{+}\,{(}^{-}\,{)}^{-}_{a}\,{(}^{+}_{b}\,{)}^{+}\,{[}^{+}_{1}\,{(}^{+}_{c}\,{)}^{+}\,{]}^{+}\,{]}^{+} \\
t &= {[}^{+}_{0}\,{]}^{+}\,{(}^{+}_{a}\,{)}^{+}\,{(}^{-}\,{)}^{-}_{a}\,{(}^{+}_{b}\,{)}^{+}\,{[}^{+}_{1}\,{]}^{+}\,{(}^{+}_{c}\,{)}^{+}
\end{aligned}$$

After the push of $c$ returns, we have $\mathtt{q[1]} = [{(}^{+}_{a}\,{)}^{+}\,{(}^{-}\,{)}^{-}_{a}\,{(}^{+}_{b}\,{)}^{+}\,{(}^{+}_{c}\,{)}^{+}]$. When the first ${]}^{+}$ occurs, the first three actions in the `q[1]` must be emitted. □

*Lemma 6.4. Given an instrumented execution of an N-`Stack`, the linearized trace of the execution is QQC with the emitted specification.*

PROOF SKETCH. Let us refer to a sequence like ${(}^{+}_{a}\,{)}^{+}\,{(}^{-}\,{)}^{-}_{a}\,{(}^{+}_{b}\,{)}^{+}$ as a *chain*. A chain is a sequence of calls that can be emitted from a single queue without any intervening change to `e`. By Theorem 5.3 suffices to show that after the execution of each atomic, the number of chains is bounded by the number of open calls. This follows by induction on the length of the instrumented execution. □

In light of Lemma 6.4, to show that the $N$-`Stack` is QQC, it suffices to show that the emitted specification is indeed a stack specification. Unfortunately, as observed in [13], this fails to hold.

*Example 6.5.* As discussed in Example 1.4, the linearized trace ${[}^{+}_{a}\,{]}^{+}\,{(}^{+}_{b}\,{)}^{+}\,{[}^{+}_{c}\,{[}^{-}\,{]}^{-}_{a}\,{]}^{+}$ generates the specification ${[}^{+}_{a}\,{]}^{+}\,{(}^{+}_{b}\,{)}^{+}\,{[}^{-}\,{]}^{-}_{a}\,{[}^{+}_{c}\,{]}^{+}$. However, this specification is not a stack trace. With some number of initial pushes, this execution is still possible: The linearized trace ${[}^{+}_{x}\,{]}^{+}\,{(}^{+}_{y}\,{)}^{+}\,{[}^{+}_{a}\,{]}^{+}\,{(}^{+}_{b}\,{)}^{+}\,{[}^{+}_{c}\,{[}^{-}\,{]}^{-}_{a}\,{]}^{+}$ generates the specification ${[}^{+}_{x}\,{]}^{+}\,{(}^{+}_{y}\,{)}^{+}\,{[}^{+}_{a}\,{]}^{+}\,{(}^{+}_{b}\,{)}^{+}\,{[}^{-}\,{]}^{-}_{a}\,{[}^{+}_{c}\,{]}^{+}$. □

This problematic execution occurs because a push and pop are racing at the first stack, yet the pop retrieves a prior value: the pop has *overtaken* the push. We must disallow such executions. It is not sufficient to require only that pop operations block on an empty stack.

*Definition 6.6.* An execution is *properly-popped* if for every push $a$ and pop $b$ that are assigned the same stack `s[i]`,

$$\mathsf{time}_1(a) < \mathsf{time}_1(b) \text{ implies } \mathsf{time}_2(a) < \mathsf{time}_2(b). \qquad \square$$

*Lemma 6.7. If an execution of the instrumented $N$-`Stack` is properly-popped, then it trace it prints is a stack trace.*

PROOF SKETCH. It is sufficient to note that the execution of the emitter follows the same pattern as the uninstrumented $N$-`Stack` on a sequential execution. (This is only true with proper popping.) The result follows since, as shown in [13], the sequential execution of the $N$-`Stack` does simulate a stack. □

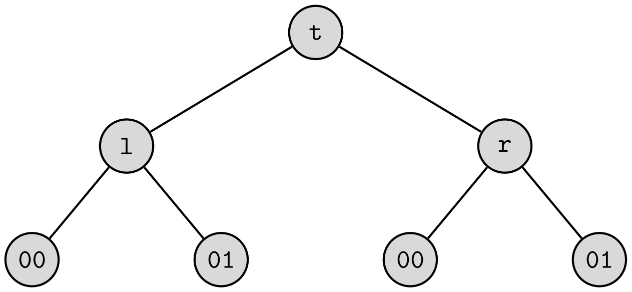
*Theorem 6.8. Any properly-popped execution of an $N$-`Stack` is QQC.*

PROOF. By Lemmas 6.4 and 6.7. □

We have shown that for properly-popped executions (where a pop may not ignore a concurrent push on the same stack) the $N$-`Stack` is QQC. As noted in the introduction, we know of no analogous condition for increment/decrement counters.

In [13], Shavit and Touitou show that in a quiescent state, their elimination-tree stack reaches a state consistent with a stack. We now consider the relation between our $N$-`Stack`s and these elimination-tree stacks.

*Example 6.9.* A depth-2 elimination-tree stack can be implemented using three atomic booleans—top (`t`), left (`l`) and right (`r`)—and 4 linearizable stacks with addresses `00`, `01`, `10` and `11`.



The *address* of a stack in an depth-$d$ elimination tree is a sequence of $d$ booleans, indicating the value of the boolean at each level, going down a branch of the tree. Both `push` and `pop` toggle the booleans as they go down the tree, using an atomic read and update. If $\texttt{t} = 0$, then `push` sets $\texttt{t} = 1$ and goes left. If $\texttt{t} = 0$, then `pop` sets $\texttt{t} = 1$ and goes right. The methods follow this same pattern down the tree until they reach the bottom-level stack, at which point they perform the operation. Initially all booleans are set to 0. For example, one uninstrumented execution proceeds as follows.

$$\begin{aligned}
&\langle \mathtt{t}=0, \langle \mathtt{l}=0, \mathtt{s}_\mathtt{l}=[[\,],[\,]]\ \rangle, \langle \mathtt{r}=0, \mathtt{s}_\mathtt{r}=[[\,],[\,]]\ \rangle\rangle\\
\xrightarrow{(^{+}_{e}}&\langle \mathtt{t}=1, \langle \mathtt{l}=0, \mathtt{s}_\mathtt{l}=[[\,],[\,]]\ \rangle, \langle \mathtt{r}=0, \mathtt{s}_\mathtt{r}=[[\,],[\,]]\ \rangle\rangle\\
\xrightarrow{[^{+}_{b}\,]^{+}}&\langle \mathtt{t}=0, \langle \mathtt{l}=0, \mathtt{s}_\mathtt{l}=[[\,],[\,]]\ \rangle, \langle \mathtt{r}=1, \mathtt{s}_\mathtt{r}=[[b],[\,]]\rangle\rangle\\
\xrightarrow{[^{+}_{a}\,]^{+}}&\langle \mathtt{t}=1, \langle \mathtt{l}=1, \mathtt{s}_\mathtt{l}=[[a],[\,]]\ \rangle, \langle \mathtt{r}=1, \mathtt{s}_\mathtt{r}=[[b],[\,]]\rangle\rangle\\
\xrightarrow{[^{+}_{d}\,]^{+}}&\langle \mathtt{t}=0, \langle \mathtt{l}=1, \mathtt{s}_\mathtt{l}=[[a],[\,]]\ \rangle, \langle \mathtt{r}=0, \mathtt{s}_\mathtt{r}=[[b],[d]]\rangle\rangle\\
\xrightarrow{[^{+}_{c}\,]^{+}}&\langle \mathtt{t}=1, \langle \mathtt{l}=0, \mathtt{s}_\mathtt{l}=[[a],[c]]\ \rangle, \langle \mathtt{r}=0, \mathtt{s}_\mathtt{r}=[[b],[d]]\rangle\rangle\\
\xrightarrow{)^{+}}&\langle \mathtt{t}=1, \langle \mathtt{l}=1, \mathtt{s}_\mathtt{l}=[[ea],[c]]\rangle, \langle \mathtt{r}=0, \mathtt{s}_\mathtt{r}=[[b],[d]]\rangle\rangle\\
\xrightarrow{\{^{-}\,\}^{-}_{e}}&\langle \mathtt{t}=0, \langle \mathtt{l}=0, \mathtt{s}_\mathtt{l}=[[a],[c]]\ \rangle, \langle \mathtt{r}=0, \mathtt{s}_\mathtt{r}=[[b],[d]]\rangle\rangle\\
\xrightarrow{\{^{-}\,\}^{-}_{d}}&\langle \mathtt{t}=1, \langle \mathtt{l}=0, \mathtt{s}_\mathtt{l}=[[a],[c]]\ \rangle, \langle \mathtt{r}=1, \mathtt{s}_\mathtt{r}=[[b],[\,]]\rangle\rangle\\
\xrightarrow{\{^{-}\,\}^{-}_{c}}&\langle \mathtt{t}=0, \langle \mathtt{l}=1, \mathtt{s}_\mathtt{l}=[[a],[\,]]\ \rangle, \langle \mathtt{r}=1, \mathtt{s}_\mathtt{r}=[[b],[\,]]\rangle\rangle\\
\xrightarrow{\{^{-}\,\}^{-}_{b}}&\langle \mathtt{t}=1, \langle \mathtt{l}=1, \mathtt{s}_\mathtt{l}=[[a],[\,]]\ \rangle, \langle \mathtt{r}=0, \mathtt{s}_\mathtt{r}=[[\,],[\,]]\ \rangle\rangle\\
\xrightarrow{\{^{-}\,\}^{-}_{a}}&\langle \mathtt{t}=0, \langle \mathtt{l}=0, \mathtt{s}_\mathtt{l}=[[\,],[\,]]\ \rangle, \langle \mathtt{r}=0, \mathtt{s}_\mathtt{r}=[[\,],[\,]]\ \rangle\rangle
\end{aligned}$$

This gives the trace $(^{+}_{e}\ [^{+}_{b}\,]^{+}\ [^{+}_{a}\,]^{+}\ [^{+}_{d}\,]^{+}\ [^{+}_{c}\,]^{+}\ )^{+}\ \{^{-}\,\}^{-}_{e}\ \{^{-}\,\}^{-}_{d}\ \{^{-}\,\}^{-}_{c}\ \{^{-}\,\}^{-}_{b}\ \{^{-}\,\}^{-}_{a}$ which is QQC with respect to $[^{+}_{a}\,]^{+}\ [^{+}_{b}\,]^{+}\ [^{+}_{c}\,]^{+}\ [^{+}_{d}\,]^{+}\ (^{+}_{e}\,)^{+}\ \{^{-}\,\}^{-}_{e}\ \{^{-}\,\}^{-}_{d}\ \{^{-}\,\}^{-}_{c}\ \{^{-}\,\}^{-}_{b}\ \{^{-}\,\}^{-}_{a}$ . Our 4-`Stack` does not generate this execution trace; however, our 2-`Stack` does. In general, our $N^d$-`Stack` has strictly fewer behaviors than the $N$-branching elimination-tree stack of depth $d$. We leave open the question of whether a $N$-branching elimination-tree stack of depth $d$ has behaviors that not possible for an $N$-`Stack`. □

The instrumented execution of a $N$-branching elimination-tree stack of depth $d > 1$ can be defined using the execution of elimination-tree stacks of depth $d - 1$, using the same strategy as our $N$-`Stack`. While the balancer's behavior is more general in the composed system, the emitter's is not: The emitter code is entirely sequentialized, therefore a 2-nested $N$-branching emitter has the same behavior as a flat $N^2$-branching emitter.

*Theorem 6.10. Any properly-popped execution of a $N$-branching elimination-tree stack of depth $d$ is QQC.*

PROOF SKETCH. Following the strategy in Theorem 6.8, we need only prove the corresponding lemmas. In each case, the proof procedes by induction on $d$. In each case the basis is the same: a depth 1 elimination tree stack is simply an $N$-`Stack`.

The analogue of Lemma 6.4 follows, as before, by induction on the length of the instrumented execution. An open call at depth $d$ may initiate a new chain, but only in *one* stack of depth $d - 1$.

For the analogue of 6.7 it suffices to observe that the emitter's behavior is the same if levels $d > 1$ and $d - 1$ are flattened into a single level of size $N^2$. This follows from the atomicity of the emitter. □

## 7 Conclusions

*Quantitative quiescent consistency (QQC)* is a correctness criterion for concurrent data structures that relaxes linearizability and refines quiescent consistency. To the best of our knowledge, it is the first such criterion to be proposed.

To show that QQC is a robust concept, we have provided three alternate characterizations: (1) in the style of linearizability, (2) counting the number of calls before a return, and (3) using speculative flat combining. We have also proven compositionality (in the style of Herlihy and Wing [9]) and the correctness of data structures defined by Aspnes, Herlihy, and Shavit [3] and Shavit and Touitou [13].

In order to establish the correctness of the elimination-tree stack of [13], we had to restrict attention to traces in which no pop *overtakes* a push on the same stack. A related constraint appears in a footnote of [11]: “To keep things simple, pop operations should block until a matching push appears.” This, however, is not strong enough to guarantee quiescent consistency as we have defined it. Our analysis provides a full account: The stack is QQC with the no-overtaking requirement and only weakly quiescently consistent without it.

As witnessed by the stack trace $\{^{+}_{c}\ [^{-}\ ]^{-}_{a}\ (^{+}_{a}\ )^{+}\ \}^{+}$ in section 5, QQC does not enforce causality. We have chosen not to treat causality in this paper in order to present the basic idea of QQC as clearly as possible. Causality is an orthogonal concept: One can enforce causality in *addition* to QQC. We have not done so here because causality requires a notion of derivation over the underlying values, in which one distinguishes public values (eg, base types) from secret values (eg, pointers or nonces). For stacks, derivation requires that the value returned by a pop must be either public or a previously pushed secret.

Linearizability has proven to be a valuable foundation for program verification techniques. It remains to be seen if QQC can be of use in this regard.

Linearizability is, at its core, *linear*. We have defined QQC in terms of general partial orders, and yet the results reported here are stated in terms of sequential specifications. Partly we have done this so that we can relate the definition of QQC to the vast amount of existing work on linearizability. However, the general case is interesting.

## Acknowledgements

Gustavo Petri participated in the early discussions which motivated this work. Alan Jeffrey and Corin Pitcher made useful comments on previous candidate definitions for QQC.

## References

[1] Y. Afek, G. Korland, and E. Yanovsky, “Quasi-linearizability: relaxed consistency for improved concurrency,” in *OPODIS*, ser. LNCS, vol. 6490, Springer, 2010, pp. 395–410.

[2] W. Aiello et al., “Supporting increment and decrement operations in balancing networks,” *Chicago J. Theor. Comput. Sci.*, 2000.

[3] J. Aspnes, M. Herlihy, and N. Shavit, “Counting networks,” *J. ACM*, vol. 41, no. 5, pp. 1020–1048, 1994.

[4] C. Busch and M. Mavronicolas, “The strength of counting networks (abstract),” in *PODC*, J. E. Burns and Y. Moses, Eds., ACM, 1996, p. 311.

[5] C. Dwork, M. Herlihy, and O. Waarts, “Contention in shared memory algorithms,” *J. ACM*, vol. 44, no. 6, pp. 779–805, 1997.

[6] A. Haas et al., “Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation,” in *Conf. Computing Frontiers*, ACM, 2013, p. 17.

[7] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, “Flat combining and the synchronization-parallelism tradeoff,” in *SPAA*, 2010, pp. 355–364.

[8] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova, “Quantitative relaxation of concurrent data structures,” in *POPL*, 2013, pp. 317–328.

[9] M. Herlihy and J. M. Wing, “Linearizability: a correctness condition for concurrent objects,” *ACM TOPLAS*, vol. 12, no. 3, pp. 463–492, 1990.

[10] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[11] N. Shavit, “Data structures in the multicore age,” *Commun. ACM*, vol. 54, no. 3, pp. 76–84, Mar. 2011.

[12] N. Shavit and D. Touitou, “Elimination trees and the construction of pools and stacks (preliminary version),” in *SPAA*, 1995, pp. 54–63.

[13] —, “Elimination trees and the construction of pools and stacks,” *Theory Comput. Syst.*, vol. 30, no. 6, pp. 645–670, 1997.