Hoare-Style Specifications as Correctness Conditions for Non-linearizable Concurrent Objects

Ilya Sergey[†] Aleksandar Nanevski[‡] Anindya Banerjee[‡] Germán Andrés Delbianco[‡]

[†]University College London, UK [‡]IMDEA Software Institute, Spain

i.sergey@ucl.ac.uk {aleks.nanevski, anindya.banerjee, german.delbianco}@imdea.org

Abstract

Designing efficient concurrent objects often requires abandoning the standard specification technique of *linearizability* in favor of more relaxed correctness conditions. However, the variety of alternatives makes it difficult to choose which condition to employ, and how to compose them when using objects specified by different conditions.

In this work, we propose a *uniform* alternative in the form of Hoare logic, which can explicitly capture—in the auxiliary state—the interference of environment threads. We demonstrate the expressiveness of our method by verifying a number of concurrent objects and their clients, which have so far been specified only by non-standard conditions of *concurrency-aware linearizability*, *quiescent*, and *quantitative quiescent consistency*. We report on the implementation of the ideas in an existing Coq-based tool, providing the first mechanized proofs for all the examples in the paper.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Theory, Verification

Keywords Concurrency, Hoare logic, linearizability, quiescent consistency, counting networks, mechanized proofs.

1. Introduction

Linearizability [25] remains the most well-known correctness condition for concurrent objects. It works by relating a concurrent object to a sequential behavior. More precisely, for each concurrent history of an object, linearizability requires that there exists a mapping to a sequential history, such that the ordering of matching call/return pairs is preserved either if they are performed by the same thread, or if they do not overlap. As such, linearizability has been used to establish the correctness of a variety of concurrent objects such as stacks, queues, sets, locks, and snapshots—all of which have intuitive sequential specs.

However, as argued by Shavit [42], efficient parallelization may require the development of concurrent objects that are inherently *non-linearizable*: in the presence of interference, such objects exhibit behavior that is not reducible to any sequential behavior via linearizability. To reason about such objects, a variety of novel conditions has been developed: concurrency-aware linearizability (CAL) [21], quiescent consistency (QC) [4, 10], quasi-linearizability (QL) [2], quantitative relaxation [23], quantitative quiescent consistency (QQC) [28], and local linearizability [20], to name a few. These conditions, formulated as relations on execution traces, specify a program's behavior under concurrent interference. Some, such as QC, devote special treatment to the sequential case, qualifying the behavior in the quiescent (*i.e.*, interference-free) moments.

This proliferation of alternative conditions is problematic, as it makes all of them non-canonical. For any specific example, it is difficult to determine which condition to use, or if a new one should be developed. Worse, each new condition requires a development of its own dedicated program logic or verification tool. Furthermore, it is unclear how to combine the conditions/logics/tools, when different ones have been used for different subprograms. Finally, having criteria defined *semantically*, *e.g.*, in terms of execution traces, makes it challenging to employ them directly for reasoning about clients of the corresponding data structures.

1.1 Concurrency Specification via Program Logics

In this paper, we propose an alternative, uniform, approach: a Hoare logic equipped with special *subjective* kind of auxiliary state [32] that makes it possible to name the amount of concurrent interference, and relate it to the program's inputs and outputs *directly*, without reducing to sequential behavior. We use Fine-grained Concurrent Separation Logic (FCSL) [35], designed to reason about higher-order concurrent programs, and has been recently implemented as a verification tool on top of Coq [40], but whose ability to address non-linearizable programs has not been observed previously.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

OOPSLA'16, October 30–November 4, 2016, Amsterdam, Netherlands.
Copyright © 2016 ACM 978-1-4503-4444-9/16/10. . . $15.00.
http://dx.doi.org/10.1145/2983990.2983999

More specifically, subjective auxiliary state permits that within a spec of a thread, one can refer to the private state (real or auxiliary) of *other* interfering threads *in a local manner*. This private state can have arbitrary user-specified structure, as long as it satisfies the properties of a partial commutative monoid (PCM). A particularly important PCM is that of *time-stamped histories*, which has previously been applied to linearizable objects [41], where it replaced call/return histories. A (logically) time-stamped history consists of entries of the form $t \mapsto a$, signifying that an atomic behavior a occurred at a time (or linearization point) t. A subjective specification further distinguishes the histories of the thread and its interfering environment, and usefully relates both to the thread's input and output.

Of course, Hoare-style reasoning about histories is a natural idea, exploited recently in several works [5, 16, 18, 22]. Here, however, we rely on the unifying power of PCMs, in combination with subjective specifications, to show that by generalizing histories in different ways—though all subject to PCM laws—we can capture the essence of several different conditions, such as CAL, QC and QQC in one-and-same *off-the-shelf* logical system and tool. More precisely, our histories need not merely identify a point at which an atomic behavior logically occurred, but can also include information about interference, or lack thereof. Moreover, we will use generic FCSL constructs for delimiting the scope of auxiliary state, to reason about quiescent moments.

1.2 Contributions and Outline

The ability to use FCSL for specifying and verifying *linearizable* objects (*e.g.*, fine-grained stacks and atomic snapshots) has been recognized before [41]. In contrast, the main conceptual contribution of this work is an observation that the *very same* abstractions provided by FCSL are sufficient to ascribe non-trivial *non-linearizable* objects with specs that can hide object implementation details, but are sufficiently strong to be used in proofs of concurrent client programs, as we demonstrate in Section 2. Specifically, we recognize that auxiliary histories can be subject of user-defined invariants beyond mere adherence to sequential executions (*e.g.*, be concurrency-aware [21]), and can be used to capture intermediate interference, allowing for quantitative reasoning about outcomes of concurrent executions (*e.g.*, in the spirit of QQC [28]). These observations, surprisingly, enabled reasoning about non-linearizable data structures and their clients, which were never previously approached from the perspective of program logics or mechanically verified.

In this unified approach based on program logic, it seems inherently impossible (and contrary to the whole idea) to classify Hoare triples as corresponding to this or that correctness condition. Thus, instead of providing theorems that relate Hoare triples to existing conditions, we justify the adequacy of our approach by proof-of-concept verifications of concurrent objects and their clients.

Hence, as key technical contributions, we present *subjective specs and the first mechanized proofs* (in Coq) of (1) an elimination-based exchanger [39] (Section 3), previously specified using CAL, and (2) a simple counting network [4] (Section 6) that inspired definitions of QC and QQC. We then employ these specs to verify client programs (Sections 5 and 7). We discuss alternative design choices for specs and further applications of our verification approach in Section 8, and summarize our mechanization experience in Section 9. Section 10 compares to related work and Section 11 concludes.

2. Main Ideas and Overview

We begin by outlining the high-level intuition of our specification approach, and summarize the main formalization steps. As the first motivating example, we consider the concurrent exchanger structure from `java.util.concurrent` [14, 39]. The main purpose of the exchanger is to allow two threads to efficiently swap values in a non-blocking way via a globally shared channel. The exchange might fail, if a thread trying to swap a value does not encounter a peer to do that in a predefined period of time.

For instance, the result of the two-thread program

$$\overset{\boxed{T_1}}{r_1 := \texttt{exchange}\ 1} \ || \ \overset{\boxed{T_2}}{r_2 := \texttt{exchange}\ 2} \tag{1}$$

can be described by the following assertion:[1]

$$r_1 = r_2 = \texttt{None} \vee r_1 = \texttt{Some}\ 2 \wedge r_2 = \texttt{Some}\ 1 \tag{2}$$

That is, r_1 and r_2 store the results of the execution of subthreads T_1 and T_2 correspondingly, and both threads either succeed, exchanging the values, or fail. The ascribed outcome is only correct under the assumption that no other threads besides T_1 and T_2 attempt to use the very same exchange channel concurrently.

Why is the exchanger not a linearizable data structure? To see that, recall that linearizability reduces the concurrent behavior to a sequential one [25]. If the exchanger were linearizable, all possible outcomes of the program (1) would be captured by the following two sequential programs, modelling selected interleavings of the threads T_1 and T_2:

$$\begin{gathered} r_1 := \texttt{exchange}\ 1;\ r_2 := \texttt{exchange}\ 2; \\ \text{and} \\ r_2 := \texttt{exchange}\ 2;\ r_1 := \texttt{exchange}\ 1; \end{gathered} \tag{3}$$

However, both programs (3) will *always* result in $r_1 = r_2 = \texttt{None}$, as, in order to succeed, a call to the exchanger needs another thread, running concurrently, with which to

[1] We use ML-style `option` data type with two constructors, `Some` and `None` to indicate success and failure of an operation, correspondingly.

exchange values. This observation demonstrates that linearizability with respect to a sequential specification is too weak a correctness criterion to capture the `exchanger`'s behavior observed in a truly concurrent context [21]: an adequate notion of correctness for `exchange` must mention the effect of interference.

Consider another structure, whose concurrent behavior cannot be related to sequential executions via linearizability:

$$\begin{array}{l}\texttt{flip2}\ (x : \texttt{ptr nat})\ :\ \texttt{nat}\ =\ \{\\ \quad a := \texttt{flip}\ x;\\ \quad b := \texttt{flip}\ x;\\ \quad \textbf{return}\ a + b\ \ \}\end{array} \tag{4}$$

The procedure `flip2` takes a pointer x, whose value is either 0 or 1 and changes its value to the opposite, twice, via the *atomic* operation `flip`, returning the sum of the previous values. Assuming that x is being modified only by the calls to `flip2`, what is the outcome r of the following program?

$$r := \texttt{flip2}\ x; \tag{5}$$

The answer depends on the presence or absence of interfering threads that invoke `flip2` concurrently with the program (5). Indeed, in the absence of interference, `flip2` will flip the value of x twice, returning the sum of 0 and 1, *i.e.*, 1. However, in the presence of other threads calling `flip2` in parallel, the value of r may vary from 0 to 2.

What are the intrinsic properties of `flip2` to be specified? Since the effect of `flip2` is distributed between two internal calls to `flip`, both subject to interference, the specification should capture that the variation in `flip2`'s result is subject to interference. Furthermore, the specification should be expressive enough to allow reasoning under bounded interference. For example, in the absence of interference from any other threads besides T_1 and T_2 that invoke `flip2` concurrently, the program below will always result in $r = 2$:

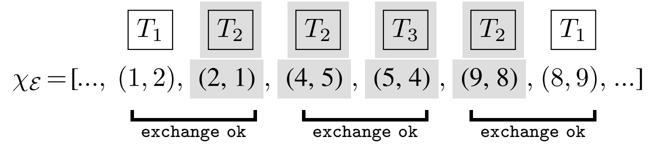
$$\begin{array}{c}\boxed{T_1} \qquad\qquad \boxed{T_2}\\ r_1 := \texttt{flip2}\ x \parallel r_2 := \texttt{flip2}\ x;\\ r := r_1 + r_2\end{array} \tag{6}$$

2.1 Abstract Histories of Non-linearizable Objects

Execution histories capture the traces of a concurrent object's interaction with various threads, and are a central notion for specifying concurrent data structures. For example, linearizability specifies the behavior of an object by mapping the object's global history of method invocations and returns to a sequence of operations that can be observed when the object is used sequentially [25]. However, as we have shown, neither `exchange` nor `flip2` can be understood in terms of sequential executions.

We propose to specify the behavior and outcome of such objects in terms of *abstract concurrent histories*, as follows. Instead of tracking method invocations and returns, our histories track the "interesting" changes to the object's state. What is "interesting" is determined by the user, depending on the intended clients of the concurrent object. Moreover, our specifications are subjective (*i.e.*, thread-relative) in the following sense. Our histories do not identify threads by their thread IDs. Instead, each method is specified by relating two different history variables: the history of the invoking thread (aka. *self*-history), and the history of its concurrent environment (aka. *other*-history). In each thread, these two variables have different values.

For example, in the case of the exchanger, the interesting changes to the object's state are the exchanges themselves. Thus, the global history $\chi_\mathcal{E}$ tracks the successful exchanges in the form of pairs of values, as shown in below:



The diagram presents the history from the viewpoint of thread T_1. The exchanges made by T_1 are colored white, determining the *self*-history of T_1. The gray parts are the exchanges made by the other threads (e.g., T_2, T_3, *etc*.), and determine the *other*-history for T_1.

The subjective division between *self* and *other* histories emphasizes that a successful exchange is actually represented by *two* pairs of numbers (x, y) and (y, x), that appear consecutively in $\chi_\mathcal{E}$, and encode the two ends of an exchange from the viewpoint of the exchanging threads. We call such pairs *twins*. As an illustration, the white entry $(2, 1)$ from the self-history of T_1, is matched by a twin gray entry $(1, 2)$ from the other-history of T_1, encoding that T_1 exchanging 2 for 1 corresponds to T_1's environment exchanging 1 for 2.

The subjective division is important, because it will enable us to specify threads *locally*, *i.e.*, without referring to the code of other threads. For example, in the case of program (1), we will specify that T_1, in the case of a successful exchange, adds a pair $(1, r_1)$ to its self history, where $\texttt{Some}\ r_1$ is T_1's return value. Similarly, T_2 adds a pair $(2, r_2)$ to its self history, where $\texttt{Some}\ r_2$ is T_2's return value.

On the other hand, it is an important invariant of the exchanger object—but not of any individual thread—that twin entries are symmetric pairs encoding different viewpoints of the one-and-the-same exchange. This object invariant will allow us to reason about clients containing combinations of exchanging threads. Taking program (1) as an example again, the object invariant will imply of the individual specifications of T_1 and T_2, that r_1 must equal 2, and r_2 must equal 1, if no threads interfered with T_1 and T_2.

We can similarly employ abstract histories to specify `flip2`. One way to do it is to notice that the value of the shared counter x will be changing as $0, 1, 0, 1, \ldots$, and exactly *two* of these values will be contributed by each call to

`flip2` made by some thread. We can depict a particular total history $\chi_\mathcal{F}$ of the `flip2` structure as follows:

$$\chi_\mathcal{F} = [\ \ldots,\ \underset{T_1}{1},\ \underset{T_2}{0},\ \underset{T_1}{1},\ \underset{T_2}{0},\ \underset{T_3}{1},\ \underset{T_3}{0},\ \ldots\ ]$$

T_1.flip2

The two "white" contributions are made by thread T_1's call to `flip2`, while the rest (gray) are contributions by T_1's environment. Since the atomic `flip` operation returns the *complementary* (*i.e.*, previous) value of the counter, the overall result of T_1's call in this case is $\bar{1} + \bar{1} = 0 + 0 = 0$.

The invariant for the `flip2` structure postulates the interleaving 0/1-shape of the history and also ensures that the last history entry is x's current value. This will allow us to reason about clients of `flip2`, such as (6). In the absence of interference, we can deduce that the two parallel calls to `flip2` have contributed four *consecutive* entries to the history $\chi_\mathcal{F}$, with each thread contributing precisely two of them. For each of the two calls, the result equals the sum of the two complementary values for what the corresponding thread has contributed to the history, hence, the overall sum $r_1 + r_2$ is 2.

2.2 Hoare-style Specifications for `exchange` and `flip2`

The above examples illustrate that subjectivity and object invariants are two sides of the same coin. In tandem, they allow us to specify threads individually, but also reason about thread combinations. We emphasize that in our approach, the invariants are *object-specific* and *provided by the user*. For example, we can associate the invariant about twin entries with the exchanger structure, but our method will not mandate the same invariant for other structures for which it is not relevant. This is in contrast to using a fixed correctness condition, such as linearizability, QC, or CAL, which cannot be parametrized by user-defined properties.[2]

Subjective histories can be encoded in our approach as *auxiliary state* [36, 41]. Our Hoare triples will specify how programs modify their histories, while the invariants are declared as properties of a chunk of shared state (*e.g.*, resource invariants of [36]). With the two components, we will be able to describe the effects and results of programs *declaratively*, *i.e.*, without exposing program implementations.

A semi-formal and partial spec of `exchange` looks as follows, with the white/gray parts denoting *self*/*other* contributions to history, from the point of view of the thread being specified (we postpone the full presentation until Section 3):

$$\begin{array}{c} \{\chi_\mathcal{E} = [\ \ldots\ ]\} \\ \texttt{exchange}\ v \\ \left\{ \begin{array}{c} \textsf{if res is Some } w \textsf{ then} \\ \chi_\mathcal{E} = [\ \ldots\ , (v, w), \ \ldots\ ] \textsf{ else } \chi_\mathcal{E} = [\ \ldots\ ] \end{array} \right\} \end{array} \tag{7}$$

The ellipsis $(\ldots)$ stands for an existentially-quantified chunk of the history. The spec (7) says that a successful exchange adds an entry (v, w) to the *self*-history (hence, the entry is white). In the case of failed exchange, no entry is added. In the complete and formal specification in Section 3, we will have to add a timing aspect, and say that the new entry appears *after* all the history entries from the precondition. We will also have to say that no entries are removed from the *other* history (*i.e.*, the exchanger cannot erase the behavior of other threads), but we elide those details here.

The spec of `flip2` is defined with respect to history $\chi_\mathcal{F}$:

$$\begin{array}{c} \{\chi_\mathcal{F} = [\ \ldots\ ]\} \\ \texttt{flip2}\ x \\ \{\ \exists a\ b, \chi_\mathcal{F} = [\ \ldots\ , a, \ \ldots\ , b, \ \ldots\ ], \textsf{res} = \bar{a} + \bar{b}\ \} \end{array} \tag{8}$$

It says that the return value res is equal to the sum of binary complements $\bar{a} + \bar{b}$ for the thread's two separate *self*-contributions to the history. Due to the effects of the interference, the history entries a and b may be separated in the overall history by the contributions of the environment, as indicated by $\ldots$ between them.

2.3 Using Subjective Specifications in the Client Code

The immediate benefit of using Hoare logic is that one can easily reason about programs whose components use different object invariants, whereas there is not much one can say about programs whose components require different correctness conditions. For example, Figure 1 shows a proof sketch for a toy program that uses both `exchange` and `flip2`. As each of these methods requires its own auxiliary history variable ($\chi_\mathcal{E}$ for the exchanger, and $\chi_\mathcal{F}$ for `flip2`), the combined program uses both, but the proof simply ignores those histories that are not relevant for any specific method (*i.e.*, we can "frame" the specs (7) and (8) wrt. the histories of the objects that they do not depend upon).

The program first forks two instances of `flip2`, storing the results in r_1 and r_2 (line 4). Next, two new threads are forked, trying to exchange r_1 and r_2 (line 8). The conditional (line 12) checks if the exchange was successful, and if so, assigns the sum of exchanged values to t (line 14); otherwise t gets assigned 2. We want to prove via the specs (7) and (8), that in the absence of external interference on the `flip2`'s pointer x and the exchanger, the outcome is always $t = 2$.

Explaining the verification In addition to the absence of external interference, we assume that the initial value of x is 0, and the initial *self*-histories for both `flip2` and `exchange` are empty (line 1). Once the `flip2` threads are forked, we employ spec (8) for each of them, simply ignoring (i.e.,

[2] For example, linearizability does not allow users to declare history invariants on a per-object basis. The exchanger example motivated the introduction of the correctness condition CAL [21], which relaxes linearizability, and makes it somewhat more general in this respect, but still falls short of admitting user-defined invariants. `flip2` can be specified using a variation of QC [28], but we show that a similar property can be expressed via subjectivity and a user-defined invariant.

$$
\begin{array}{ll}
1 & \{\chi_{\mathcal{F}} = \emptyset, \chi_{\mathcal{E}} = \emptyset\} \\
2 & \{\chi_{\mathcal{F}} = [\ \dots\ ]\} \quad \Big\| \quad \{\chi_{\mathcal{F}} = [\ \dots\ ]\} \\
3 & r_1 := \texttt{flip2}\ x \quad \Big\| \quad r_2 := \texttt{flip2}\ x \\
4 & \{\ \exists a\, b, \chi_{\mathcal{F}} = [\ \dots\ , a, \ \dots\ , b, \ \dots\ ], r_1 := \bar{a} + \bar{b}\ \} \quad \Big\| \quad \{\ \exists c\, d, \chi_{\mathcal{F}} = [\ \dots\ , c, \ \dots\ , d, \ \dots\ ], r_2 := \bar{c} + \bar{d}\ \} \\
5 & \{\chi_{\mathcal{F}} = \mathsf{perm}(a, b, c, d) = [1, 0, 1, 0], r_1 = \bar{a} + \bar{b}, r_2 = \bar{c} + \bar{d}\} \\
6 & \{r_1 + r_2 = 2\} \\
7 & \{\chi_{\mathcal{E}} = [\ \dots\ ]\} \quad \Big\| \quad \{\chi_{\mathcal{E}} = [\ \dots\ ]\} \\
8 & s_1 := \texttt{exchange}\ r_1 \quad \Big\| \quad s_2 := \texttt{exchange}\ r_2 \\
9 & \left\{ \begin{array}{l} \mathsf{if}\ s_1\ \mathsf{is\ Some}\ v_1\ \mathsf{then} \\ \chi_{\mathcal{E}} = [\ \dots\ , (r_1, v_1), \ \dots\ ]\ \mathsf{else}\ \chi_{\mathcal{E}} = [\ \dots\ ] \end{array} \right\} \quad \Big\| \quad \left\{ \begin{array}{l} \mathsf{if}\ s_2\ \mathsf{is\ Some}\ v_2\ \mathsf{then} \\ \chi_{\mathcal{E}} = [\ \dots\ , (r_2, v_2), \ \dots\ ]\ \mathsf{else}\ \chi_{\mathcal{E}} = [\ \dots\ ] \end{array} \right\} \\
10 & \{s_1 = \mathsf{Some}\ v_2 \wedge s_2 = \mathsf{Some}\ v_2 \implies \chi_{\mathcal{E}} = \mathsf{perm}((r_1, v_1), (r_2, v_2)) = \mathsf{perm}((v_1, r_1), (v_2, r_2))\} \\
11 & \{s_1 = \mathsf{Some}\ v_2 \wedge s_2 = \mathsf{Some}\ v_2 \implies v_1 = r_2 \wedge v_2 = r_1\} \\
12 & \textbf{if}\ s_1\ \textbf{is}\ \texttt{Some}\ v_1\ \textbf{and}\ s_2\ \textbf{is}\ \texttt{Some}\ v_2\ \textbf{then} \\
13 & \{v_1 = r_2, v_2 = r_1, r_1 + r_2 = 2\} \\
14 & t := v_1 + v_2 \quad \{t = 2\} \quad \textbf{else}\ t := 2 \quad \{t = 2\}
\end{array}
$$

Figure 1. Verification of a concurrent client program using `exchange` and `flip2` in the absence of external interference.

framing out) $\chi_{\mathcal{E}}$, as this history variable does not apply to them `flip2`. Upon finishing, the postconditions of `flip2` in line 4 capture the relationship between the contributions to the history $\chi_{\mathcal{F}}$ and the results r_1 and r_2 of the two calls.

Both postconditions in line 4 talk about the very same history $\chi_{\mathcal{F}}$, just using different colors to express that the contributions of the two threads are *disjoint*: a and b being white in the left thread, implies that a and b are history entries added by the left thread. Thus, they *must* be gray in the right thread, as they cannot overlap with the entries contributed by the right thread. The right thread cannot explicitly specify in its postcondition that a and b are gray, since the right thread is unaware of the specific contributions of the left thread.

Dually, c and d being white in the right thread in line 4, implies that they must be gray on the left. Thus, overall, in line 5, we know that $\chi_{\mathcal{F}}$ contains all four entries in some permutation, and in the absence of intereference, it contains no other entries but these four. From the object invariant on $\chi_{\mathcal{F}}$ it then follows that the entries are some permutation of $[1, 0, 1, 0]$, which makes their sum total $r_1 + r_2 = 2$.

Similarly, we ignore $\chi_{\mathcal{F}}$ while reasoning about calls to `exchange` via spec (7) (lines 7 and 9). As before, we know that the entry (r_1, v_1), which is white in the left postcondition in line 9, must be gray on the right, and dually for (r_2, v_2). In total, the history $\chi_{\mathcal{E}}$ must contain both of the entries, but, by the invariant, it must also contain their twins. In the absence of any other interference, it therefore must be that (r_1, v_1) is a twin for (r_2, v_2), *i.e.*, $r_1 = v_2$ and $r_2 = v_1$, as line 11 expresses for the case of a succesful exchange. The rest of the proof is then trivial.

The sketch relied on several important aspects of program verification in FCSL: *(i)* the invariants constraining $\chi_{\mathcal{F}}$ and $\chi_{\mathcal{E}}$ were preserved by the methods, *(ii)* upon joining the threads, we can rely on the disjointness of history contributions of the two threads, in order to combine the thread-local views into a specification of the parent thread, and, *(iii)* we could guarantee the absence of the external interference.

The aspect *(i)* is a significant component of what it means to specify and verify a concurrent object. As we will show in Sections 3 and 6, defining a sufficiently strong object invariant, and then proving that it is indeed an invariant, *i.e.*, that it is preserved by the implementation of the program, is a major part of the verification challenge. We will explain FCSL rules for *parallel composition* and *hiding* in Section 4, justifying the reasoning principles *(ii)* and *(iii)*.

2.4 Specifying Non-linearizable Objects in Three Steps

As shown by Sections 2.1–2.3, our method for specifying and verifying non-linearizable concurrent objects and their clients boils down to the following three systematic steps.

Step 1 (§2.1): *Define object-specific auxiliary state and its invariants. The auxiliary state will typically include a specific notion of abstract histories, recording whatever behavior is perceived as essential by the implementor of the object.* To account for the variety of object-specific correctness conditions, we do not fix a specific shape for the histories. We do not restrict them to always record pairs of numbers (as in the exchanger), or record single numbers (as in `flip2`). The only requirement that we impose on auxiliary state in general, and on histories in particular, is that the chosen type of auxiliary state is an instance of the PCM algebraic structure [41], thus providing an abstract, and user-defined, notion of *disjointness* between *self*/*other* contributions.

Step 2 (§2.2): *Formulate Hoare-style specifications, parametrized by interference, and verify them.* This step provides a suitable "interface" for the methods of the concur-

rent object, which the clients use to reason, without knowing the details of the object and method implementations. Naturally, the interface can refer to the auxiliary state and histories defined in the previous step. When dealing with non-linearizable objects in FCSL, it is customary to formulate the spec in a subjective way (*i.e.*, using *self*/*other*, dually white/gray division between history entries) so that the specification has a way to refer to the effects of the interfering calls to the same object. The amount of interference can be later instantiated with more specific information, once we know more about the context of concurrent threads in which the specified program is being run.

Step 3 (§2.3): *Restrict the interference when using object specs for verification of clients.* Eventually, thread-local knowledge about effects of individual clients of one and the same object, should be combined into a cumulative knowledge about the effect of the composition. To measure this effect, one usually considers the object in a *quiescent* (interference-free) moment [38]. To model quiescent situations, FCSL provides a program-level constructor for *hiding*. In particular, `hide` e executes e, but statically prevents other threads from interfering with e, by making e's auxiliary history invisible. Program e's *other* contribution is fixed to be empty, thus modeling quiescence.

3. Verifying the Exchanger Implementation

We now proceed with more rigorous development of the invariants and specification for the exchanger data structure, necessary to verify its real-world implementation [14], which was so far elided from the overview of the approach.

The exchanger implementation is presented in ML-style pseudo-code in Figure 2. It takes a value $v : A$ and creates an *offer* from it (line 2). An offer is a pointer p to two consecutive locations in the heap.[3] The first location stores v, and the second is a "hole" which the interfering thread tries to fill with a matching value. The hole is drawn from the type $\mathtt{hole} = \mathsf{U} \mid \mathsf{R} \mid \mathsf{M}\ w$. Constructor U signals that the offer is unmatched; R that the exchanger retired (*i.e.*, withdrew) the offer, and does not expect any matches on it; and $\mathsf{M}\ w$ that the offer has been matched with a value w.

The global pointer g stores the latest offer proposed for matching. The exchanger proposes p for matching by making g point to p via the atomic compare-and-set instruction `CAS` (line 3). We assume that `CAS` returns the value read, which can be used to determine if it failed or succeeded. If `CAS` succeeds, exchanger waits a bit, then checks if the offer has been matched by some w (lines 6, 7). If so, `Some` w is returned (line 7). Otherwise, the offer is retired by storing R into its hole (line 6). Retired offers remain allocated (thus, exchanger has a memory leak) in order to avoid the ABA problem, as usual in many concurrent structures [24, 46]. If the exchanger fails to link p into g in line 3, it deallocates

[3] In our mechanization, we simplify a bit by making p point to a pair instead.

```
1   exchange (v : A) : option A = {
2     p ← alloc (v, U);
3     b ← CAS (g, null, p);
4     if b == null then
5       sleep (50);
6       x ← CAS (p+1, U, R);
7       if x is M w then return (Some w)
8       else return None
9     else
10      dealloc p;
11      cur ← read g;
12      if cur == null then return None
13      else
14        x ← CAS (cur+1, U, M v);
15        CAS (g, cur, null);
16        if x is U then w ← read cur; return (Some w)
17        else return None}
```

Figure 2. Elimination-based exchanger procedure.

the offer p (line 10), and instead tries to match the offer cur that is current in g. If no offer is current, perhaps because another thread already matched the offer that made the `CAS` in line 3 fail, the exchanger returns None (line 12). Otherwise, the exchanger tries to make a match, by changing the hole of cur into $\mathsf{M}\ v$ (line 14). If successful (line 16), it reads the value w stored in cur that was initially proposed for matching, and returns it. In any case, it unlinks cur from g (line 15) to make space for other offers.

3.1 Step 1: Defining Auxiliary State and Invariants

To formally specify the exchanger, we decorate it with auxiliary state. In addition to histories, necessary for specifying the observable behavior, the auxiliary state is used for capturing the coherence constraints of the actual implementation, *e.g.*, with respect to memory allocation and management of outstanding offers. The state is subjective as described in Section 2: it keeps thread-local auxiliary variables that name the thread's private state (*self*), but also the private state of all other threads combined (*other*).

The subjective state of the exchanger for each thread in this example consists of three groups of two components: (1) thread-private heap h_S of the thread, and of the environment h_O, (2) a set of outstanding offers π_S created by the thread, and by the environment π_O, and (3) a time-stamped history of values χ_S that the thread exchanged so far, and dually χ_O for the environment. In Section 2, we illustrated subjectivity by means of histories, white we used white and gray entries, respectively, to describe what here we name χ_S and χ_O, respectively. Now we see that the dichotomy extends beyond histories, and this example requires the dichotomy applied to heaps, and to sets of offers as well. In addition to *self*/*other* components of heaps, permissions and histories, we also need shared (aka. *joint*) state consisting of two components: a heap h_J of storing the offers that have been made, and

a map m_J of offers that have been matched, but not yet collected by the thread that made them.

Heaps, sets and histories are all PCMs under the operation of disjoint union, with empty heap/set/history as a unit. We overload the notation and write $x \mapsto v$ for a singleton heap with a pointer x storing value v, and $t \mapsto a$ for a singleton history. Similarly, we apply disjoint union $\cupdot$ and subset $\subseteq$, to all three types uniformly.

We next describe how the exchanger manipulates the above variables. First, h_J is a heap that serves as the "staging" area for the offers. It includes the global pointer g. Whenever a thread wants to make an offer, it allocates a pointer p in h_S, and then tries to move p from h_S into h_J, simultaneously linking g to p, via the `CAS` in line 3 of Figure 2.

Second, π_S and π_O are sets of offers (hence, sets of pointers) that determine offer ownership. A thread that has the offer $p \in \pi_\mathsf{S}$ is the one that created it, and thus has the *sole* right to retire p, or to collect the value that p was matched with. Upon collection or retirement, p is removed from π_S.

Third, χ_S and χ_O are exchanger-specific histories, each mapping a time-stamp (isomorphic to nats), to a pair of exchanged values. A singleton history $t \mapsto (v, w)$ symbolizes that a thread having this singleton as a subcomponent of χ_S, has exchanged v for w at time t. As we describe below, the most important invariant of the exchanger is that each such singleton is matched by a "symmetric" one to capture that another thread has *simultaneously* exchanged w for v. Classical linearizability cannot express this simultaneous behavior, making the exchanger non-linearizable.

Fourth, m_J is a map storing the offers that were matched, but not yet acknowledged and collected. Thus, $\mathsf{dom}\ m_\mathsf{J} = \pi_\mathsf{S} \cupdot \pi_\mathsf{O}$. A singleton entry in m_J has the form $p \mapsto (t, v, w)$ and denotes that offer p, initially storing v, was matched at time t with w. A singleton entry is entered into m_J when a thread on the one end of matching, matches v with w. Such a thread also places the *twin* entry $\bar{t} \mapsto (w, v)$, with inverted order of v and w, into its own private history χ_S, where:

$$\bar{t} = \begin{cases} t+1 & \text{if } t \text{ is odd} \\ t-1 & \text{if } t > 0 \text{ and } t \text{ is even} \end{cases}$$

For technical reasons, 0 is not a valid time-stamp, and has no distinct twin. The pending entry for p resides in m_J until the thread that created the offer p decides to "collect" it. It removes p from m_J, and simultaneously adds the entry $t \mapsto (v, w)$ into its own χ_S, thereby logically completing the exchange. Since twin time-stamps are consecutive integers, a history cannot contain entries *between* twins.

Thus, two twin entries in the combined history including χ_S, χ_O and m_J, jointly represent a single exchange, as if it occurred *atomically*. For example, the entries $1 \mapsto (v_1, w_1)$ and $2 \mapsto (w_1, v_1)$ will encode the end-points of the first exchange; the entries $3 \mapsto (v_2, w_2)$ and $4 \mapsto (w_2, v_2)$ will encode the end-points of the second exchange, etc., the entries at timestamps t and $t+1$, for odd t, will encode the end-points of the $\frac{t+1}{2}$-th exchange. *Concurrency-aware* histories [21] capture this by making the ends of an exchange occur as simultaneous events. We capture it via twin time-stamps. More formally, consider $\chi = \chi_\mathsf{S} \cupdot \chi_\mathsf{O} \cupdot \|m_\mathsf{J}\|$. Then, the exchanger's main invariant is that χ always contains matching twin entries:

$$t \mapsto (v, w) \subseteq \chi \iff \bar{t} \mapsto (w, v) \subseteq \chi \qquad (9)$$

Here $\|m_\mathsf{J}\|$ is the collection of all the entries in m_J. That is, $\|\emptyset\| = \emptyset$, and $\|p \mapsto (t, v, w) \cupdot m_\mathsf{J}'\| = t \mapsto (v, w) \cupdot \|m_\mathsf{J}'\|$.

In our implementation, we prove that atomic actions, such as `CAS`, preserve the invariant, therefore, the whole program, being just a composition of actions, doesn't violate it.

3.2 Step 2: Hoare-style Specification of Exchanger

We can now give the desired formal Hoare-style spec.

$$\begin{gathered} \{h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \emptyset, \chi_\mathsf{S} = \emptyset, \eta \subseteq \chi_\mathsf{O} \cupdot \|m_\mathsf{J}\|\} \\ \mathtt{exchange}\ v \\ \left\{ \begin{array}{c} h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \emptyset, \eta \subseteq \chi_\mathsf{O} \cupdot \|m_\mathsf{J}\|, \\ \text{if res is Some } w \text{ then} \\ \exists t.\ \chi_\mathsf{S} = t \mapsto (v, w), \mathsf{last}(\eta) < t, \bar{t} \text{ else } \chi_\mathsf{S} = \emptyset \end{array} \right\} \end{gathered} \qquad (10)$$

The precondition says that the exchanger starts with the empty private heap h_S, set of offers π_S and history χ_S; hence by framing, it can start with any value for these components.[4] The logical variable η names the initial history of all threads, $\chi_\mathsf{O} \cupdot \|m_\mathsf{J}\|$, which may grow during the call, thus, we use subset instead of equality to make the precondition stable under other threads adding entries to χ_O or m_J.

In the postcondition, the self heap h_S and the set of offers π_S didn't change. Hence, if `exchange` made an offer during its execution, it also collected or retired it by the end. The history η is still a subset of the ending value for $\chi_\mathsf{O} \cupdot \|m_\mathsf{J}\|$, signifying that the environment history only grows by interference. We will make a crucial use of this part of the spec when verifying a client of the exchanger in Section 5.

If the exchange fails (*i.e.*, res is None), then χ_S remains empty. If it succeeds (either in line 7 or line 16 in Figure 2), *i.e.*, if the result res is Some w, then there exists a time-stamp t, such that self-history χ_S contains the entry $t \mapsto (v, w)$, symbolizing that v and w were exchanged at time t.

Importantly, the postcondition implies, by invariant (9), that in the success case, the twin entry $\bar{t} \mapsto (w, v)$ must belong to $\chi_\mathsf{O} \cupdot \|m_\mathsf{J}\|$, *i.e.*, *another* thread matched the exchange (this was made explicit by the spec (7)). Moreover, the exchange occurred *after* the call to exchange: whichever η we chose in the pre-state, both t and $\bar{t}$ are larger than the last time-stamp in η.

The proof outline for the exchanger is available in Appendix A. In Section 5, after introducing necessary FCSL background, we will illustrate ***Step 3*** of our method and show how to employ the subjective Hoare spec (10) for modular verification of a concurrent client.

[4] Framing in FCSL is similar to that of separation logic, allowing extensions to the initial state that remain invariant by program execution. In FCSL, however, framing applies to any PCM-valued state component (*e.g.*, heaps, histories, *etc.*), whereas in separation logic, it applies just to heaps.

4. Background on FCSL

In order to formally present ***Step 3*** of our method, we first need to introduce some important parts of FCSL.

A Hoare specification in FCSL has the form $\{P\}\ e\ \{Q\}@\mathcal{R}$. P and Q are pre- and postcondition for partial correctness, and $\mathcal{R}$ defines the *shared resource* on which e operates. The latter is a state transition system describing the invariants of the state (real and auxiliary) and atomic operations that can be invoked by the threads that simultaneously operate on that state. We elide the transition system aspect of resources here, and refer to [35] for detailed treatment.

An important secondary role of a resource is to declare the variables that P and Q may scope over. For example, in the case of exchanger, we use the variables $h_{\mathsf{S}}, \pi_{\mathsf{S}}, \chi_{\mathsf{S}}$, $h_{\mathsf{O}}, \pi_{\mathsf{O}}, \chi_{\mathsf{O}}$, and $h_{\mathsf{J}}, m_{\mathsf{J}}$. The mechanism by which the variables are declared is as follows. Underneath, a resource comes with only three variables: a_{S}, a_{O} and a_{J} standing for abstract self state, other state, and shared (joint) state, but the user can pick their types depending on the application. In the case of exchanger, a_{S} and a_{O} are triples containing a heap, an offer-set and a history. The variables we used in Section 2 are projections out of such triples: $a_{\mathsf{S}} = (h_{\mathsf{S}}, \pi_{\mathsf{S}}, \chi_{\mathsf{S}})$, and $a_{\mathsf{O}} = (h_{\mathsf{O}}, \pi_{\mathsf{O}}, \chi_{\mathsf{O}})$. Similarly, $a_{\mathsf{J}} = (h_{\mathsf{J}}, m_{\mathsf{J}})$.

It is essential that a_{S} and a_{O} have a common type exhibiting the algebraic structure of a PCM, under a partial binary operation $\mathbin{\dot\cup}$. PCMs give a way, generic in $\mathcal{R}$, to define the inference rule for parallel composition.

$$\frac{\{P_1\}\ e_1\ \{Q_1\}@\mathcal{R} \quad \{P_2\}\ e_2\ \{Q_2\}@\mathcal{R}}{\{P_1 \circledast P_2\}\ e_1 \parallel e_2\ \{[\mathsf{res}.1/\mathsf{res}]Q_1 \circledast [\mathsf{res}.2/\mathsf{res}]Q_2\}@\mathcal{R}} \quad (11)$$

Here, $\circledast$ is defined as follows.

$$(P_1 \circledast P_2)(a_{\mathsf{S}}, a_{\mathsf{J}}, a_{\mathsf{O}}) \iff \exists x_1\ x_2.\, a_{\mathsf{S}} = x_1 \mathbin{\dot\cup} x_2, \\ P_1(x_1, a_{\mathsf{J}}, x_2 \mathbin{\dot\cup} a_{\mathsf{O}}), P_2(x_2, a_{\mathsf{J}}, x_1 \mathbin{\dot\cup} a_{\mathsf{O}})$$

Thereby, when a parent thread forks e_1 and e_2, then e_1 becomes part of the environment for e_2, and vice-versa. This is so because the *self* component a_{S} of the parent is split into x_1 and x_2; x_1 becomes the *self* part of e_1, but x_2 is added to the *other* part a_{O} of e_1 (and symmetrically for e_2).

To reason about quiescent moments, we use one more constructor of FCSL: *hiding*. The program hide e operationally executes e, but logically installs a resource within the scope of e. In the case of the exchanger, hide e starts only with private heaps h_{S} and h_{O}, then takes a chunk of heap out of h_{S} and "installs" an exchanger in this heap, allowing the threads in e to exchange values. hide e is *quiescent* wrt. exchanger, as the typechecker will prevent composing hide e with threads that want to exchange values with e.

The auxiliaries $\pi_{\mathsf{S}}, \chi_{\mathsf{S}}$, $\pi_{\mathsf{O}}, \chi_{\mathsf{O}}$, and $h_{\mathsf{J}}, m_{\mathsf{J}}$, belonging to the exchanger (denoted as resource $\mathcal{E}$) are visible within hide, but outside, only h_{S} persists (denoted as a resource $\mathcal{P}$ for private state). We elide the general hiding rule [35], and just show the special case for the exchanger.

$$\frac{\{P\}\ e\ \{Q\}@\mathcal{E}}{\{h_{\mathsf{S}} = \Phi_1(h_{\mathsf{J}}), \Phi_1(P)\}\ \mathsf{hide}\ e\ \{\exists \Phi_2.\, h_{\mathsf{S}} = \Phi_2(h_{\mathsf{J}}), \Phi_2(Q)\}@\mathcal{P}} \quad (12)$$

Read bottom-up, the rule says that we can install the exchanger $\mathcal{E}$ in the scope of a thread that works with $\mathcal{P}$, but then we need substitutions Φ_1 and Φ_2, to map variables of $\mathcal{E}$ ($h_{\mathsf{S}}, \pi_{\mathsf{S}}, \chi_{\mathsf{S}}$, *etc*) to values expressed with variables from $\mathcal{P}$ (h_{S} and h_{O}). Φ_1 is an initial such substitution (user provided), and the rule guarantees the existence of an ending substitution Φ_2. The substitutions have to satisfy a number of side conditions, which we elide here for brevity. The most important one is that *other* variable $a_{\mathsf{O}} = (h_{\mathsf{O}}, \pi_{\mathsf{O}}, \chi_{\mathsf{O}})$ is fixed to be the PCM unit (*i.e.*, a triple of empty sets). Fixing a_{O} to unit captures that hide protects e from interference.

At the beginning of hide e, the private heap equals the value that Φ_1 gives to h_{J} ($h_{\mathsf{S}} = \Phi_1(h_{\mathsf{J}})$). In other words, the hide rule takes the private heap of a thread, and makes it shared, *i.e.*, gives it to the h_{J} component of $\mathcal{E}$. Upon finishing, hide e makes h_{J} private again.

In the subsequent text we elide the resources from specs.

5. Verifying Exchanger's Client

We next illustrate how the formally specified exchanger from Section 3 can be used by real-world client programs, and how the *other* component, asserted by the spec to satisfy $\eta \subseteq \chi_{\mathsf{O}} \mathbin{\dot\cup} \|m_{\mathsf{J}}\|$, is crucial for their verification. We emphasize that the proof of the client does not see the implementation details, which are hidden by the spec (10).

While simple, our client is realistic, and has been used in `java.util.concurrent` [14]. It is defined as follows. First, the exchanger loops until it exchanges the value.

```
exchange' (v : A) : A = {
  w' ← exchange v;
  if w' is Some w then return w else exchange' v }
```

Next, `exchange'` is iterated to exchange a sequence in order, appending the received matches to an accumulator.

```
ex_seq (vs, ac : seq A) : seq A = {
  if vs is v::vs' then
    w ← exchange' v;  ex_seq (vs', snoc ac w)
  else return ac }
```

Our goal is to prove, via (10), that the parallel composition

$$e = \mathtt{ex_seq}\ (vs_1, \mathtt{nil}) \parallel \mathtt{ex_seq}\ (vs_2, \mathtt{nil})$$

exchanges vs_1 and vs_2, *i.e.*, returns the pair (vs_2, vs_1). This holds only under the assumption that e runs without interference (*i.e.*, quiescently), so that the two threads in e have no choice but to exchange the values between themselves.

We make the quiescence assumption explicit using the FCSL hide constructor, as described in Section 4. Thus, we

establish the following Hoare triple:

$$\{h_\mathsf{S} = g \mapsto \mathsf{null}\}\ \mathsf{hide}\ e\ \{g \in \mathsf{dom}\ h_\mathsf{S}, \mathsf{res} = (vs_2, vs_1)\} \tag{13}$$

It says that we start with a heap where g stores null, and end with a possibly larger heap (due to the memory leak), but with the result (vs_2, vs_1). The auxiliaries $\pi_\mathsf{S}, \pi_\mathsf{O}, \eta_\mathsf{S}, \eta_\mathsf{O}, h_\mathsf{J}, m_\mathsf{J}$ are visible inside hide, but outside, only h_S persists.

Explaining the verification. We illustrate the verification by listing the specs of selected subprograms. First, the spec of `exchange'` easily derives from (10) by removing the now-impossible failing case.

$$\{h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \emptyset, \chi_\mathsf{S} = \emptyset, \eta \subseteq \chi_\mathsf{O} \uplus \|m_\mathsf{J}\|\}$$
$$\texttt{exchange'}\ v$$
$$\left\{\begin{array}{c} h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \emptyset, \eta \subseteq \chi_\mathsf{O} \uplus \|m_\mathsf{J}\|, \\ \exists t.\, \chi_\mathsf{S} = t \mapsto (v, \mathsf{res}), \mathsf{last}(\eta) < t, \bar{t} \end{array}\right\}$$

Next, `ex_seq` has the following spec:

$$\{h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \emptyset, \chi_\mathsf{S} = \emptyset\}$$
$$\mathsf{ex_seq}\ (vs, \mathsf{nil})$$
$$\left\{\begin{array}{c} \exists ts.\, h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \emptyset, \chi_\mathsf{S} = \mathsf{zip}\ ts\ vs\ \mathsf{res}, \\ \mathsf{grows_notwins}\ ts, \mathsf{zip}\ \overline{ts}\ \mathsf{res}\ vs \subseteq \chi_\mathsf{O} \uplus \|m_\mathsf{J}\| \end{array}\right\}$$

Here, ts is a list of time-stamps, and zip $ts\ vs\ ws$ joins up the singleton histories $t \mapsto (v, w)$, for each t, v, w drawn, in order, from the lists ts, vs, ws. The spec says that at the time-stamps from ts, `ex_seq` exchanged the elements of vs for those of `res`. That ts is increasing and contains no twins, follows from the spec of `exchange'` which says that the time-stamps t and $\bar{t}$ that populate ts and $\overline{ts}$, are larger than anything in η, and thus only grow with iteration. From the same postcondition, it follows that $\chi_\mathsf{O} \uplus \|m_\mathsf{J}\|$ contains all the twin exchanges, by invariant (9), as commented in Section 2 about the spec for `exchange`.

Next, by the FCSL parallel composition rule (Section 4):

$$\{h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \emptyset, \chi_\mathsf{S} = \emptyset\}$$
$$\mathsf{ex_seq}\ (vs_1, \mathsf{nil}) \parallel \mathsf{ex_seq}\ (vs_2, \mathsf{nil})$$
$$\left\{\begin{array}{c} \exists ts_1\ ts_2.\, \mathsf{grows_notwins}\ ts_1, \mathsf{grows_notwins}\ ts_2, \\ h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \emptyset, \chi_\mathsf{S} = \mathsf{zip}\ ts_1\ vs_1\ \mathsf{res}.1 \uplus \mathsf{zip}\ ts_2\ vs_2\ \mathsf{res}.2, \\ \mathsf{zip}\ \overline{ts_1}\ \mathsf{res}.1\ vs_1 \subseteq \mathsf{zip}\ ts_2\ vs_2\ \mathsf{res}.2 \uplus \chi_\mathsf{O} \uplus \|m_\mathsf{J}\|, \\ \mathsf{zip}\ \overline{ts_2}\ \mathsf{res}.2\ vs_2 \subseteq \mathsf{zip}\ ts_1\ vs_1\ \mathsf{res}.1 \uplus \chi_\mathsf{O} \uplus \|m_\mathsf{J}\|. \end{array}\right\}$$

To explain: ts and `res` from the left and right `ex_seq` threads become ts_1, ts_2, `res`.1 and `res`.2, respectively. The values of each *self* component h_S, π_S, χ_S from the two threads are joined into the *self* component of the composition. At the same time, the *other* component χ_O of the left (resp. right) thread equals the sum of χ_S of the right (resp. left) thread, and the χ_O of the composition. This formalizes the intuition that upon forking, the left thread becomes part of the environment for the right thread, and vice-versa.

The postcondition says that the self history of e contains both zip $ts_1\ vs_1$ res.1 and zip $ts_2\ vs_2$ res.2. Thus, vs_1 is exchanged for res.1, and vs_2 for res.2. But we further want

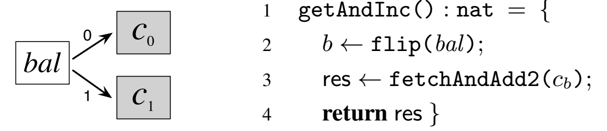


Figure 3. Simple counting network

to derive res.1 $= vs_2$ and res.2 $= vs_1$, *i.e.*, the lists are exchanged *for each other*, in the absence of interference.

We next explain how this desired property follows for hide e, from the two inequalities in e's postcondition

$$\mathsf{zip}\ \overline{ts_1}\ \mathsf{res}.1\ vs_1 \subseteq \mathsf{zip}\ ts_2\ vs_2\ \mathsf{res}.2 \uplus \chi_\mathsf{O} \uplus \|m_\mathsf{J}\|, \tag{14}$$
$$\mathsf{zip}\ \overline{ts_2}\ \mathsf{res}.2\ vs_2 \subseteq \mathsf{zip}\ ts_1\ vs_1\ \mathsf{res}.1 \uplus \chi_\mathsf{O} \uplus \|m_\mathsf{J}\|. \tag{15}$$

Notice that (14) and (15) are ultimately instances of the conjunct $\eta \subseteq \chi_\mathsf{O} \uplus \|m_\mathsf{J}\|$ that was part of the specification (10), thereby justifying the use of subjective *other* variables.

We know that dom $m_\mathsf{J} = \pi_\mathsf{S} \uplus \pi_\mathsf{O}$ (from Section 2), that $\pi_\mathsf{S} = \emptyset$ (from e's postcondition), and that by hiding, $\pi_\mathsf{O} = \chi_\mathsf{O} = \emptyset$. Thus, towards deriving the postcondition of hide e, we simplify (14) and (15) into:

$$\mathsf{zip}\ \overline{ts_1}\ \mathsf{res}.1\ vs_1 \subseteq \mathsf{zip}\ ts_2\ vs_2\ \mathsf{res}.2$$
$$\mathsf{zip}\ \overline{ts_2}\ \mathsf{res}.2\ vs_2 \subseteq \mathsf{zip}\ ts_1\ vs_1\ \mathsf{res}.1$$

Because ts_1 and ts_2 are increasing lists of time-stamps, and contain no twins, the above implies $ts_2 = \overline{ts_1}$. Hence:

$$\mathsf{zip}\ \overline{ts_1}\ \mathsf{res}.1\ vs_1 = \mathsf{zip}\ ts_2\ vs_2\ \mathsf{res}.2$$

and thus res.1 $= vs_2$, $vs_1 =$ res.2. We omit the remaining technical argument that explains how the heap h_J, with the pointer g, is folded into h_S, which ultimately obtains (13).

6. Specifying Counting Networks

We now show how to use subjective histories to specify another class of non-linearizable objects—*counting networks*. Counting networks are a special case of *balancing networks* introduced by Aspnes *et al.* [4], themselves building on sorting networks [3], aimed to implement concurrent counters in a way free from synchronization bottlenecks. The key idea is to decompose the workload between *several* counters, so that each of them is responsible for a disjoint set of values. A thread trying to increment first approaches the *balancer*, which is a logical "switch" that "directs" the thread, *i.e.*, provides it with the address of the counter to increment. The balancers make counting networks' operations *non-linearizable*, as in the presence of interference the results of increments might be observed out of order.

Figure 3 presents a schematic outline and a pseudo-code implementation of a counting network with a single balancer. The implementation contains three pointers: the balancer *bal*, which stores either 0 or 1, thus directing

threads to the shared pointers c_0 or c_1, which count the even and odd values, respectively. Threads increment by calling getAndInc, which works as follows. It first atomically changes the bit value of the balancer via a call to atomic operation flip (line 2). The flip operation returns the *previous* value b of the balancer as a result, thus determining which of the counters, c_0 or c_1, should be incremented. The thread proceeds to atomically add 2 to the value of c_b via fetchAndAdd2 (line 3). The old value of c_b is returned as the result of the procedure.[5]

Assuming that c_0 and c_1 are initialized with 0 and 1, it is easy to see that in a single-threaded program, the network will behave as a conventional counter; that is, consecutive invocations of getAndInc return consecutive nats. However, in the concurrent setting, getAndInc may return results out of order, as follows.

Example 6.1. Consider two threads, T_1 and T_2 operating on the network initialized with $bal \mapsto 0$, $c_b \mapsto b$. T_1 calls getAndInc and executes its line 2 to set bal to 1. It gets suspended, so T_2 proceeds to execute lines 2 and 3, therefore setting bal back to 0 and returning 1. While T_1 is still suspended, T_2 calls getAndInc again, gets directed to c_0, and returns 0, after it has just returned 1.

This out-of-order behavior, however, is not random, and can be precisely characterized as a function of the number of threads operating on the network [2, 28]. In the rest of this section and in Section 7, we show how to capture such bounds in the spec using auxiliary state of (subjective) histories in a client-sensitive manner. As a form of road map, we list the desired requirements for the spec of getAndInc, adapting the design goals of the criteria, such as QC, QQC and QL [2, 4, 28], which we will proceed to verify formally, following ***Step 1*** and ***Step 2*** of our approach, and then employ in client-side reasoning via ***Step 3***:

- **R1:** Two different calls to getAndInc should return distinct results (*strong concurrent counter semantics*).
- **R2:** The results of calls to getAndInc, separated by a period of quiescence (*i.e.*, absence of interference), should appear in their sequential order (*quiescent consistency*).
- **R3:** The results of two sequential calls C_1 and C_2, in a single thread should be out of order by no more than $2\,N$, where N is the number of interfering calls that overlap with C_1 and C_2 (*quantitative quiescent consistency*).

6.1 Step 1: Counting Network's Histories and Invariants

To formalize the necessary invariants, we elaborate the counting network with auxiliary state: *tokens* (isomorphic to nats) and novel *interference-capturing histories*.

[5] In the counting network from Figure 3, the balancer itself might seem like a contention point. However, the flip operation is much less expensive than CAS as a synchronization mechanism. The performance can be further improved by constructing a *diffracting tree* of several balancers [24, §12.6], but we do not consider diffracting trees here.

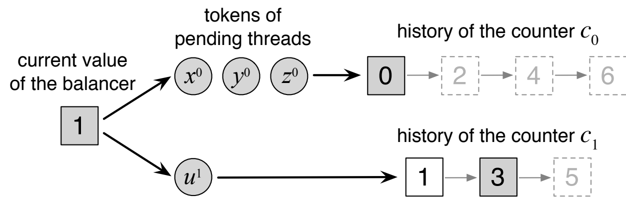


Figure 4. Tokens and histories of the simple network

A *token* provides a thread that owns it with the right to increment an appropriate counter [4]. In our example, a thread that performs the flip in line 2 of getAndInc will be awarded a token which it can then spend to execute fetchAndAdd2. Thus, any individual token represents a "pending" call to getAndInc, and the set of unspent tokens serves as a bound on the out-of-order behavior that the network exhibits. We introduce auxiliary variables for the held tokens: τ_S keeps the tokens owned by the *self* thread, with its *even* and *odd* projections τ_S^0 and τ_S^1, such that $\tau_S = \tau_S^0 \cup \tau_S^1$, administering access to c_0 and c_1, respectively. Similarly, τ_O, featuring the same projections, keeps the tokens owned by the *other* thread. We abbreviate $\tau^i = \tau_S^i \cup \tau_O^i$ for $i = 0, 1$.

Figure 4 illustrates a network with three *even* tokens: $x^0, y^0, z^0 \in \tau^0$, held by threads that will increment c_0, and one *odd* token $u^1 \in \tau^1$, whose owner will increment c_1.

A *history* of the counting network is an auxiliary finite map, consisting of entries of the form $t \mapsto (\hat{\imath}, z)$. Such an entry records that the value t has been written into an appropriate counter (c_0 or c_1, depending on the parity of t), at the moment when τ^0 and τ^1 held values of $\hat{\imath}$'s even/odd projections $\hat{\imath}^0$ and $\hat{\imath}^1$, respectively. Moreover, in order to write t into a counter, the token z was spent by the thread. We will refer to z as the *spent* token. Notice that the entries in the history contain tokens held by both *self* and *other* threads. Thus, a history captures the behavior of a thread subjectively, *i.e.*, as a function of the interfering threads' behavior.

Similarly to tokens, network histories are represented by the auxiliary variables χ_S, tracking counter updates (even and odd) performed by the *self* thread, and dually χ_O for the *other* thread. We abbreviate $\chi^i = \chi_S^i \cup \chi_O^i$ for $i = 0, 1$.

Figure 4 illustrates a moment in network's history and how it relates to the state of the counters. Only 0 has been written to c_0 so far (upon initialization), hence χ^0 only contains an entry for $t = 0$ (we ignore at the moment the *contents* of the history entries). On the other hand, χ^1 has entries for 1 and 3, because after initialization, one thread has increased c_1. The gray boxes indicate that 0 and 3 are the current values of c_0 and c_1, and thus also the latest entries in χ^0 and χ^1, respectively. In particular, these values will be returned by the next invocations of fetchAndAdd2. The dashed boxes correspond to the entries to be contributed by the currently running threads holding tokens x^0, y^0, z^0, u^1.

In addition to τ and χ which come in flavors private to *self* and *other* threads, we require the following shared variables:

(1) h_J for the joint heap of the network, and (2) b_J, n^0_J and n^1_J for the contents of bal, c_o and c_1, respectively.

Invariants of the counting network The main invariant of the network relates the number of tokens, the size of histories and the value of the balancer:

$$|\chi^0| + |\tau^0| = |\chi^1| + |\tau^1| + b_\mathsf{J} \tag{16}$$

The equation formalizes the intuition that out-of-order anomalies of the counting network appear if one of the two counters is too far ahead of the other one. The invariant (16) provides a bound on such a situation. One counter can get ahead temporarily, but then there must be a number of threads waiting to spend their tokens on the other counter. Thus, the other counter will eventually catch up.

The approaches such as quiescent and quantitative quiescent consistency describe this situation by referring to the number of *unmatched* call events in an event history [10, 28]. In contrast, we formalize this property via auxiliary state: the sets of tokens $\hat{\iota}$ recorded in the entry for the number t determine the environment's capability to add new history entries, and thus "run ahead" or "catch up" after t has been returned. The other invariants of the counting network are as follows:

(i) $h_\mathsf{J} = bal \mapsto b_\mathsf{J} \uplus c_0 \mapsto n^0_\mathsf{J} \uplus c_1 \mapsto n^1_\mathsf{J}$.

(ii) The histories contain disjoint time-stamps.

(iii) The history χ^0 (resp. χ^1) contains *all* even (resp. odd) values in $[0, n^0_\mathsf{J}]$ (resp. $[1, n^1_\mathsf{J}]$). This ensures that n^0_J and n^1_J are the last time-stamps in χ^0 and χ^1, respectively.

(iv) τ^0, τ^1 and $\mathsf{spent}\ (\chi_\mathsf{S} \uplus \chi_\mathsf{O})$ contain mutually disjoint tokens, where $\mathsf{spent}\ (t \mapsto (\hat{\iota}, z) \uplus \chi') = \{z\} \uplus \mathsf{spent}\ \chi'$, and $\mathsf{spent}\ \emptyset = \emptyset$. In other words, a spent token never appears among the "alive" ones (*i.e.*, in $\tau^0 \uplus \tau^1$).

(v) $t \mapsto (\hat{\iota}, z) \subseteq \chi_\mathsf{S} \uplus \chi_\mathsf{O} \implies z \in \hat{\iota}$.

(vi) For any t, $\hat{\iota}$, z:

- $t \mapsto (\hat{\iota}, z) \subseteq \chi^0 \implies t{+}2\,|\hat{\iota}{\cap}\tau^0| < n^1_\mathsf{J}{+}2\,|\hat{\iota}{\cap}\tau^1|{+}2$, and
- $t \mapsto (\hat{\iota}, z) \subseteq \chi^1 \implies t + 2\,|\hat{\iota} \cap \tau^1| < n^0_\mathsf{J} + 2\,|\hat{\iota} \cap \tau^0| + 2$.

The invariant (vi) provides quantitative information about the network history by relating the actual (n^0_J, n^1_J) and the past (t) counter values, via the current amount of interference (τ) and the snapshot interference ($\hat{\iota}$). To explain (vi), we resort to the intuition provided by the following equality, which, however, being *not quite valid*, cannot be used as an invariant, as we shall see. Focusing on the first clause in (vi), if $t \mapsto (\hat{\iota}, z) \subseteq \chi^0$, then, intuitively:

$$t + 2\,|\hat{\iota}^0 \setminus \tau^0| + 2\,|\hat{\iota} \cap \tau^0| = n^1_\mathsf{J} + 2\,|\hat{\iota} \cap \tau^1| + (2b_\mathsf{J} - 1)$$

The equality says the following. When t is snapshot from c_0 and placed into the history χ^0, the set of outstanding even tokens was $\hat{\iota}^0$. By the present time, c_0 has been increased $|\hat{\iota}^0 \setminus \tau^0|$ times, each time by 2, thus $n^0_\mathsf{J} = t{+}2\,|\hat{\iota}^0 \setminus \tau^0|$. What is left to add to c_0 to reach the *period of quiescence*, when no threads interfere with us, is $2\,|\hat{\iota} \cap \tau^0|$. Similar reasoning applies to c_1. It is easy to see at the period of quiescence, c_0 and c_1 differ by $2b_\mathsf{J} - 1$; that is, the counter pointed to by bal is behind by 1. However, the equality is invalid, as b_J can be read off only in the present, whereas the "intuitive" reasoning behind the equality requires a value of b_J from a quiescent period *in the future*. Hence, in order to get a valid property, we bound $2b_\mathsf{J} - 1$ by 2. For simplicity, we even further weaken the bounds by dropping $|\hat{\iota}^0 \setminus \tau^0|$ to obtain (vi); as it will turn out, even such a simpler bound will suffice for proving **R1–R3**.

Allowed changes in the counting network The state of the counting network (auxiliary and real) can be changed in two possible ways by concurrent threads. These changes formalize the way the atomic operations `flip` and `fetchAndAdd2` from Figure 3 (b) work with auxiliary state. *Flipping* alters the bit value b_J of bal to the complementary one, $1 - b_\mathsf{J}$. It also generates a token z (of parity b_J) and stores it into τ_S. The token is fresh, *i.e.*, distinct from all alive and spent tokens in $\tau_\mathsf{S} \uplus \tau_\mathsf{O} \uplus \mathsf{spent}\ (\chi_\mathsf{S} \uplus \chi_\mathsf{O})$. *Incrementation* spends a token z from τ_S, and depending on its i, it atomically increases the value n^i_J of c_i by two, while simultaneously removing z from τ_S (thus, the precondition is that $z \in \tau_\mathsf{S}$). It also adds the entry $(n^i_\mathsf{J} + 2) \mapsto (\tau^0 \uplus \tau^1, z^i)$ to χ_S, thus snapshoting the values of τ^0 and τ^1. It is easy to check that both these allowed changes preserve the state-space invariants (16), (i)–(vi), and that their effect on real state (with auxiliary state erased) are those of `flip` and `fetchAndAdd2`.

6.2 Step 2: a Hoare Spec for `getAndInc`

$$\left\{\begin{array}{c} \tau_\mathsf{S} = \emptyset, \chi_\mathsf{S} = \eta_\mathsf{S}, \eta_\mathsf{O} \subseteq \chi_\mathsf{O}, \\ \iota_\mathsf{O} \subseteq \tau_\mathsf{O} \uplus (\mathsf{spent}\ \chi_\mathsf{O} \setminus \mathsf{spent}\ \eta_\mathsf{O}), \mathcal{I}\ \eta_\mathsf{O}\ \iota_\mathsf{O} \end{array}\right\}$$

$$\texttt{getAndInc()}$$

$$\left\{\begin{array}{c} \exists \hat{\iota}\ z.\ \tau_\mathsf{S} = \emptyset, \chi_\mathsf{S} = \eta_\mathsf{S} \uplus (\mathsf{res} + 2) \mapsto (\hat{\iota}, z), \\ \eta_\mathsf{O} \subseteq \chi_\mathsf{O}, \iota_\mathsf{O} \subseteq \tau_\mathsf{O} \uplus (\mathsf{spent}\ \chi_\mathsf{O} \setminus \mathsf{spent}\ \eta_\mathsf{O}), \\ \mathsf{last}\ (\eta_\mathsf{S} \uplus \eta_\mathsf{O}) < \mathsf{res} + 2 + 2\,|\hat{\iota} \cap \iota_\mathsf{O}|, \\ \mathsf{ResPast}\ (\eta_\mathsf{S} \uplus \eta_\mathsf{O})\ \mathsf{res}\ \hat{\iota}\ z, \mathcal{I}\ \eta_\mathsf{O}\ \iota_\mathsf{O} \end{array}\right\} \tag{17}$$

Figure 5. Hoare-style spec of a simple counting network.

Figure 5 provides a Hoare-style spec for `getAndInc`, verified in our proof scripts. We use the logical variable ι and its variants to range over token sets, and η to range over histories.

The precondition starts with an empty token set ($\tau_\mathsf{S} = \emptyset$), and hence by framing, any set of tokens. The initial self-history χ_S is set to an arbitrary η_S.[6] The precondition records the *other* components of the initial state as follows. First, η_O names (a subset of) χ_O, to make it stable under interference, as in Section 2. Next, we use ι_O to name the (subset of) initially live tokens τ_O. However, as τ_O may shrink due to other threads spending tokens, simply writing $\iota_\mathsf{O} \subseteq \tau_\mathsf{O}$ is unstable. Instead, we write $\iota_\mathsf{O} \subseteq \tau_\mathsf{O} \uplus (\mathsf{spent}\ \chi_\mathsf{O} \setminus$

[6] Alternatively, we could have also taken $\chi_\mathsf{S} = \emptyset$, but the clients will require generalizing to $\chi_\mathsf{S} = \eta_\mathsf{S}$ by the FCSL's frame rule [41]. To save space and simplify the discussion, we immediately frame wrt. the auxiliary χ_S. Our examples do not require such client-side framing wrt. τ_S.

spent η_O) to account for the tokens spent by other threads as well. The set $\tau_O \uplus (\text{spent } \chi_O \setminus \text{spent } \eta_O)$ only grows under interference, as new live tokens are generated, or old live tokens are spent, making the inclusion of ι_O stable. Indeed, one cannot take *any* arbitrary η_O and ι_O to name the *other* components of the initial state. Therefore, we constrain these two variables by the invariant $\mathcal{I}$, that relates them to the *self*-components of the actual state and to each other according to the invariants (ii)–(vi).[7] This is natural, since, as we will see in Section 7, all clients instantiate η_O and ι_O with the *other*-components of the actual pre-state, respecting (ii)–(vi).

The postcondition asserts that the final token set τ_S is also empty (*i.e.*, the token that `getAndInc` generates by `flip`, is spent by the end). The history χ_S is increased by an entry $(\text{res} + 2) \mapsto (\hat{\iota}, z)$, corresponding to writing the value of the result (plus two) into one of the network's counters, snapshoting the tokens of that moment into $\hat{\iota}$, and spending the token z on the write. η_O is a subset of the new value of χ_O, and ι_O is a subset of the new value of $\tau_O \uplus (\text{spent } \chi_O \setminus \text{spent } \eta_O)$, by the already discussed stability.

The next inequality describes where the entry for $\text{res} + 2$ is placed wrt. the pre-state history $\eta = \eta_S \uplus \eta_O$. η may have gaps arising due to out-of-order behavior of the network, and $\text{res} + 2$ may fill one such gap. However, there is a bound on how far res (and hence $\text{res}+2$) may be from the tail of η. We express it as a function of ι_O and $\hat{\iota}$, derived from the bounds in (vi), taking $\text{res}+2$ for t and over-approximating the instant value n^i_J of the incremented counter via $\text{last } (\eta_S \uplus \eta_O)$. The inequality weakens the invariant (vi), making it hold for even and odd entries by moving $2\,|\hat{\iota} \cap \iota^i_O|$ (for $i = 0, 1$) to the right side of $<$ and joining them, since $\iota^0_O \cap \iota^1_O = \emptyset$.

Finally, the predicate ResPast provides more bounds that we will need in the proofs of the client code's properties.

$$\begin{aligned} &\text{ResPast } \eta \text{ res } \hat{\iota}\, z \;\hat{=}\; \hat{\iota} \subseteq \tau_O \uplus (\text{spent } \chi_O) \uplus \{z\}\,, \\ &\quad \forall t\, \iota.\, t \mapsto (\iota, -) \subseteq \eta \Rightarrow z \notin \iota,\; t < \text{res} + 2 + 2\,(|\hat{\iota} \cap \iota|) \end{aligned} \tag{18}$$

When instantiated with $\eta = \eta_S \uplus \eta_O$, ResPast says the following. The token set $\hat{\iota}$ snapshot when $\text{res}+2$ was committed to history, is a subset of all the tokens in post-state, including the live ones (τ_O), and spent ones (spent $\chi_O \uplus \{z\}$). Moreover, if t is an entry in η, with contents $(\iota, -)$, then: (1) $z \notin \iota$, because z is a token generated when `getAndInc` executed `flip`. Hence, z is fresh wrt. any token-set from the pre-state history η; and (2) t and ι satisfy the same bounds wrt. $\text{res}+2$, as those described for the last history entry and ι_O.

How will the spec* (17) *be used? The clause $\chi_S = \eta_S \uplus (\text{res} + 2) \mapsto -$ of (17), in conjunction with the invariant (ii), ensures that any two calls to `getAndInc`, sequential or concurrent, yield different history entries, and hence different results. This establishes **R1**, which we will not discuss further.

The inequality on $\text{last } (\eta_S \uplus \eta_O)$ will provide for **R2** in client reasoning. To see how, consider the particular case when ι_O is empty, *i.e.*, the pre-state is quiescent. In that case, the intersection with $\hat{\iota}$ is empty, and we can infer that $\text{res}+2$, is larger than either counter's value in the pre-state. As we shall see in Section 7, this captures the essence of QC.

Finally, the predicate ResPast (18) establishes a bound for the "out-of-order" discrepancy between the result res and any value t committed to the history *in the past*, via $2\,|\hat{\iota} \cap \iota|$. We will further bound this value using the size of $\hat{\iota}$, and the inclusion $\hat{\iota} \subseteq \tau_O \uplus \text{spent } \chi_O$ from (18). These bounds will ultimately enable us to derive the requirement **R3**.

[7] That is, η_O and ι_O take the role of χ_O and τ_O in invariants (ii)–(vi), with $n^i_J = \text{last } (\chi_S \uplus \eta_O)^i$. The formal definition of $\mathcal{I}$ is in our proof scripts.

$$\left\{ \begin{array}{c} \tau_S = \emptyset, \chi_S = \eta_S, \eta_O \subseteq \chi_O, \mathcal{I}\, \eta_O\, \iota_O, \\ \iota_O \subseteq \tau_O \uplus (\text{spent } \chi_O \setminus \text{spent } \eta_O) \end{array} \right\}$$

$$\texttt{getAndInc()} \;\|\; e_i$$

$$\left\{ \begin{array}{c} \exists \hat{\iota}\, \eta_i.\, \tau_S = \emptyset, \chi_S = \boxed{\eta_S \uplus \eta_i \uplus (\text{res.1} + 2) \mapsto (\hat{\iota}, -)}\,, \\ \eta_O \subseteq \chi_O, \iota_O \subseteq \tau_O \uplus (\text{spent } \chi_O \setminus \text{spent } \eta_O), \mathcal{I}\, \eta_O\, \iota_O, \\ \text{last } (\eta_S \uplus \eta_O) < \boxed{\text{res.1}} + 2 + 2\,|\hat{\iota} \cap \iota_O| \end{array} \right\}$$

Figure 6. Parallel composition of `getAndInc` and e_i in (20).

7. Verifying Counting Network's Clients

Following ***Step 3*** of our verification method, we now illustrate requirements **R2** and **R3** from the previous section via two different clients which execute two sequential calls to `getAndInc`. Both clients are higher-order, *i.e.*, they are parametrized by subprograms, which can be "plugged in". The first client will exhibit a quiescence between the two calls, and we will prove that the call results appear in order, as required by **R2**. The second client will experience interference of a program with a N concurrent calls to `getAndInc`, and we will derive a bound on the results in terms of N, as required by **R3**.

Both our examples will rely on the general mechanism of hiding, presented in Section 4, as a way to logically restrict the interference on a concurrent object, in this case, a counting network, in a lexically-scoped way. To "initialize" the counting network data structure, we provide the starting values for the shared heap (h_0) and for the history (η_0), assuming that the initial set of tokens is empty:

$$\begin{aligned} h_0 &\;\hat{=}\; bal \mapsto 0 \uplus c_0 \mapsto 0 \uplus c_1 \mapsto 1 \\ \eta_0 &\;\hat{=}\; \{0 \mapsto (\{0\}, 0), 1 \mapsto (\{1\}, 1)\} \end{aligned} \tag{19}$$

That is, η_0 provides the "default" history for the initial values 0 and 1 of c_0 and c_1, with the corresponding tokens represented by numbers 0 and 1. As always with hiding, the postcondition of the hidden program will imply that τ_O and χ_O are both empty, as there is no interference at the end.

7.1 Exercising Quiescent Consistency

Our first client is the following program e_{qc}:

$$\begin{array}{ll} 1 & (\mathsf{res}_1, -) \leftarrow (\texttt{getAndInc()} \mid\mid e_1); \\ 2 & (\mathsf{res}_2, -) \leftarrow (\texttt{getAndInc()} \mid\mid e_2); \\ 3 & \textbf{return}\ (\mathsf{res}_1, \mathsf{res}_2) \end{array} \tag{20}$$

Each of the calls to getAndInc interferes with either e_1 or e_2, but in the absence of *external* interference, the quiescent state is reached between the lines 1 and 2. Hence, after executing hide e_{qc}, it should be $\mathsf{res}_1 < \mathsf{res}_2$, following **R2**.

The programs e_1 and e_2 can invoke getAndInc and modify the counters concurrently with the two calls of e_{qc}, which we capture by giving both the following generic spec:

$$\begin{array}{c} \{\ \chi_S = \emptyset, \tau_S = \emptyset, \iota \subseteq \tau_O \cup \mathsf{spent}\ \chi_O\ \} \\ e_i \\ \{\exists \eta_i.\ \chi_S = \eta_i, \tau_S = \emptyset, \iota \subseteq \tau_O \cup \mathsf{spent}\ \chi_O\} \end{array} \tag{21}$$

The postcondition allows for a number of increments via calls to getAndInc, which is reflected in the addition η_i to χ_S. However, all such calls are required to be *finished* by the end of e_i ($\tau_S = \emptyset$). As customary by now, we use the logical variable ι to name the initial set of *other* tokens.

Figure 6 provides a spec for each of the parallel compositions in the program (20), proved via the corresponding FCSL inference rule for parallel composition (11). The spec is very similar to (17) with the differences highlighted via gray boxes: (a) the self-history χ_S is increased by e_i's contribution η_i in addition to the entry, introduced by getAndInc, (b) the result of the parallel composition is a pair, but we only constrain its first component res.1, resulting from the left subprogram. We also drop the last conjunct with ResPast from (17), which we won't require for this example.

Next, we use the spec from Figure 6 to specify and verify the program e_{qc}, so far *assuming* external interference.

$$\{\ \text{Fig. 6's precondition with } \eta_S := \eta_0, \eta_O := \chi_O, \text{and } \iota_O := \tau_O\ \}\ //\ P$$
$$(\mathsf{res}_1, -) \leftarrow (\texttt{getAndInc()} \mid\mid e_1);$$
$$\left\{\begin{array}{c} \exists \eta_1.\ \tau_S = \emptyset, \chi_S = \eta'_S,\ \ldots \\ \text{where } \boxed{\eta'_S = \eta_0 \uplus \eta_1 \uplus (\mathsf{res}_1 + 2) \mapsto -}\,, \eta_O := \chi_O \text{ and } \iota_O := \tau_O \end{array}\right\}$$
$$(\mathsf{res}_2, -) \leftarrow (\texttt{getAndInc()} \mid\mid e_2);$$
$$\left\{\begin{array}{l} \exists \eta_1\ \eta_2\ \hat{\iota}.\ \tau_S = \emptyset,\ \boxed{\iota_O \subseteq \tau_O \cup (\mathsf{spent}\ \chi_O \setminus \mathsf{spent}\ \eta_O)}\,, \\ \boxed{\mathsf{last}\ (\eta'_S \uplus \eta_O) < \mathsf{res}_2 + 2 + 2\,|\hat{\iota} \cap \iota_O|}\,,\ \ldots \end{array}\right\}\ //\ Q$$
$$\textbf{return}\ (\mathsf{res}_1, \mathsf{res}_2);\ //\ =: \mathsf{res}$$
$$\{\ Q(\mathsf{res}.1/\mathsf{res}_1, \mathsf{res}.2/\mathsf{res}_2)\ \}$$

We start by instantiating the logical variables η_S, η_O and ι_O from Figure 6 with η_0, *current* χ_O and τ_O, respectively, naming the obtained precondition P. In the following assertion we focus on the clauses constraining τ_S and χ_S. To verify the second call, we instantiate η_S, η_O and ι_O from Figure 6 with $\eta'_S = \eta_0 \uplus \eta_1 \uplus (\mathsf{res}_1 + 2) \mapsto -$, *current* χ_O and τ_O, correspondingly, obtaining the postcondition, which we name Q.

$$\{\ (17)\text{'s precondition with } \eta_S := \eta_0, \eta_O := \chi_O, \text{and } \iota_O := \tau_O\ \}$$
$$\mathsf{res}_1 \leftarrow \texttt{getAndInc()};$$
$$\left\{\begin{array}{c} \exists \iota.\ \tau_S = \emptyset, \chi_S = \eta'_S, \ldots \\ \text{where } \boxed{\eta'_S = \eta_0 \uplus (\mathsf{res}_1 + 2) \mapsto (\iota, -)} \end{array}\right\}$$
$$\mathsf{res}_2 \leftarrow \texttt{getAndInc()};$$
$$\{\exists \hat{\iota}\ z.\ \mathsf{ResPast}(\eta'_S \uplus \eta_O)\ \mathsf{res}_2\ \hat{\iota}\ z, \ldots\}$$
$$\left\{\begin{array}{l} \exists \hat{\iota}\ z.\ \boxed{\hat{\iota} \subseteq \tau_O \uplus (\mathsf{spent}\ \chi_O) \uplus \{z\}}\,, z \notin \iota, \\ \quad \mathsf{res}_1 + 2 < \mathsf{res}_2 + 2 + 2\,|\hat{\iota} \cap \iota| \end{array}\right\}$$
$$\textbf{return}\ (\mathsf{res}_1, \mathsf{res}_2)\ //\ =: \mathsf{res}$$
$$\{\mathsf{res}.1 < \mathsf{res}.2 + 2\,|\tau_O \uplus \mathsf{spent}\ \chi_O|\}$$

Figure 7. Proof outline of sequential composition in (24).

The inequality in the postcondition Q gives the boundary on the out-of-order position of res_2 with respect to the *last* value in the history captured in between the two parallel compositions. The boundary is given via the size of intersection of the two sets of tokens: snapshot ($\hat{\iota}$) and "alive" between the calls (ι_O). Now, to ensure the absence of external interference, we consider the program (hide e_{qc}). By the general property of hiding (Section 4), we know that at the final state there is no interference, hence $\tau_O = \emptyset$ and $\chi_O = \emptyset$ in Q. Therefore, from the set inclusion on ι_O in Q (the grayed part), we deduce that $\iota_O = \emptyset$. As a consequence, the intersection $\hat{\iota} \cap \iota_O = \emptyset$, so from the inequality we obtain

$$\mathsf{last}\ (\eta'_S \uplus \eta_O) < \mathsf{res}.2 + 2 \tag{22}$$

But η'_S is defined as $(\mathsf{res}.1{+}2) \mapsto - \uplus \ldots$, hence, $\mathsf{res}.1{+}2 \in \mathsf{dom}\ \eta'_S$, and thus $\mathsf{res}.1 + 2 \leq \mathsf{last}\ \eta'_S$. Even more:

$$\mathsf{res}.1 + 2 \leq \mathsf{last}\ (\eta'_S \uplus \eta_O). \tag{23}$$

From (22) and (23) follows the result **R2**: $\mathsf{res}.1 < \mathsf{res}.2$.

7.2 Proving Quantitative Bounds

We next show how the spec (17) also obtains quantitative bounds on the out-of-order anomalies in terms of a number of running threads in the following program e_{qqc}:

$$\left(\begin{array}{ll} 1 & \mathsf{res}_1 \leftarrow \texttt{getAndInc()}; \\ 2 & \mathsf{res}_2 \leftarrow \texttt{getAndInc()}; \\ 3 & \textbf{return}\ (\mathsf{res}_1, \mathsf{res}_2) \end{array}\right) \mid\mid\ e \tag{24}$$

The e's spec says that the *number* of calls to getAndInc in e (*i.e.*, the size of interference e exhibits) is some fixed N:

$$\{\tau_S = \emptyset, \chi_S = \eta_S\}\ e\ \{\exists \eta.\ \tau_S = \emptyset, \chi_S = \eta_S \uplus \eta, |\eta| = N\} \tag{25}$$

Our goal is to prove that in the absence of external interference for e_{qqc}, $\mathsf{res}_1 < \mathsf{res}_2 + 2\,N$ (requirement **R3**).

We first verify the sequential composition of the two calls in (24); the proof outline is in Figure 7. As previously, we

$$\{\ \tau_S = \emptyset, \chi_S = \eta_0, \dots\ \}\ //\ P$$

$$\{\tau_S = \emptyset, \chi_S = \eta_0\}$$

```
res1 ← getAndInc();
res2 ← getAndInc();
return (res1, res2) // =: res
```

$$\{\text{res.1} < \text{res.2} + 2\,|\tau_O \uplus \text{spent}\ \chi_O|\}$$

$$\Big\|\quad \{\tau_S = \emptyset, \chi_S = \emptyset\}\quad e \quad \{\exists \eta.\, \chi_S = \eta, |\eta| = N, \dots\}$$

$$//\ \text{res}_1 := \text{res.1.1}, \text{res}_2 := \text{res.1.2}$$

$$\{\text{res}_1 < \text{res}_2 + 2\,|\tau_O \uplus \text{spent}\ (\chi_O \uplus \eta)|\}$$

$$\{\text{res}_1 < \text{res}_2 + 2\,|\tau_O \uplus \text{spent}\ \chi_O| + 2\,N\}\ //\ Q$$

Figure 8. Proof outline for the e_{qqc} program.

start by instantiating the logical variables η_S, η_O and ι_O from spec (17) with η_S, χ_O and τ_O, respectively. In the assertion, resulting by of the first `getAndInc`, we keep only the clauses involving τ_S and χ_S, dropping the rest. To verify the second `getAndInc` call, we instantiate η_S, η_O and ι_O with $\eta'_S = \eta_S \uplus (\text{res}_1 + 2) \mapsto (\iota, -)$, current χ_O and τ_O.

In the postcondition of the second call to `getAndInc`, we focus on the ResPast $(\eta'_S \uplus \eta_O)$ res_2 $\hat{\iota}$ z clause, where $\hat{\iota}$ is the set of tokens snapshot when contributing $\text{res}_2 + 2$. Unfolding the definition of ResPast from (18), we obtain $\hat{\iota} \subseteq \tau_O \uplus \text{spent}\ \chi_O \uplus \{z\}$. Also, using $(\text{res}_1 + 2) \mapsto (\iota, -)$ in the implication that the unfolding obtains, we get $z \notin \iota$ and

$$\text{res}_1 + 2 < \text{res}_2 + 2 + 2\,|\hat{\iota} \cap \iota| \tag{26}$$

Now we use the following trivial fact to simplify.

Lemma 7.1. *If $z \in \hat{\iota}$ and $z \notin \iota$, then $|\hat{\iota} \cap \iota| \leq |\hat{\iota}| - 1$.*

Using the invariant (v), Lemma 7.1 derives $|\hat{\iota} \cap \iota| \leq |\hat{\iota}| - 1$ after which, the inclusion $\hat{\iota} \subseteq \tau_O \uplus \text{spent}\ \chi_O \uplus \{z\}$ leads to

$$|\hat{\iota} \cap \iota| \leq |\tau_O \uplus \text{spent}\ \chi_O| \tag{27}$$

Combined with (26), this gives us $\text{res}_1 < \text{res}_2 + 2\,|\tau_O \uplus \text{spent}\ \chi_O|$, as shown in Figure 7's postcondition. In words, it asserts that the discrepancy between res.1 and res.2 is bounded by the size of the tokens, which are either held by the interfering threads at the end or are spent.

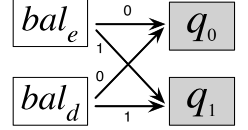
Figure 8 shows the proof outline for e_{qqc} via the spec from Figure 7. By the parallel composition rule (11), the precondition splits into two subjective views, where we send the initial history η_0 to the left thread, and the empty history to the right thread. The proof from Figure 7 then applies to the left thread, and the spec (25) applies to the right one. Final χ_O of the left thread is the union of χ_O from the joined thread with η, since the environment of the left thread includes the right thread and of the join. Rewriting by this property in the postcondition of the left thread gives us the post of the joint thread: $\text{res}_1 < \text{res}_2 + 2\,|\tau_O \uplus \text{spent}\ (\chi_O \uplus \eta)|$, which we can next simplify into

$$\text{res}_1 < \text{res}_2 + 2\,|\tau_O \uplus \text{spent}\ \chi_O| + 2\,N$$

because spent distributes over $\uplus$, and $|\text{spent}\ \eta| = |\eta| = N$. Finally, we restrict the external interference by considering (hide e_{qqc}). From the properties of hiding, we deduce that τ_O and χ_O in Q are empty, hence we can simplify into $\text{res}_1 < \text{res}_2 + 2\,N$, which is the desired result **R3**.

8. Discussion

Reasoning about quantitatively quiescent queues The idea of interference-capturing histories, which allowed us to characterize the out-of-order discrepancies between the results of a counting network in Section 6, can be applied to specify other balancer-based data structures, for instance, queues [10]. The picture on the right illustrates schematically a non-linearizable queue [10], which is built out of two *atomic* queues, q_0 and q_1, and two balancers, bal_e and bal_d. The balancers are used to distribute the workload between the two queues by directing the threads willing to enqueue and dequeue elements, correspondingly.



One can think of representing the pending enqueue/dequeue requests to each of the two queues, q_0 and q_1, by two separate sets of tokens, as shown in Figure 9. The white and gray boxes correspond to the present and dequeued nodes in the queue in the order they were added/removed. Therefore, white elements are those that are currently in the queue. Similarly, the white-colored tokens are for enqueueing elements, so the elements x, y, z and k are going to be added to the corresponding atomic queues. Gray-colored tokens correspond to dequeueing capabilities for one or another atomic queue, distributed among the threads, so the elements c and d are going to be removed next, on the expense of the corresponding dequeue tokens. The timestamps of the entries in the queue history, omitted from the figure, are created, as elements are being enqueued to q_0 and q_1, and the parity of a timestamp corresponds to the atomic queue being changed. Thus, there might be "gaps" in the combined queue history reminiscent to the gaps in the counter history from Section 6 (*e.g.*, the gap caused by the absence of an "even" element in the combined history right between d and e in Figure 9, as indicated by "?"), which will cause out-of-order anomalies during concurrent executions. By accounting for the number of past and present tokens for enqueueing and dequeueing, one should be able to capture the effects of interference and express a quantitative boundary on the discrepancy between the results, coming out of order.

How much information to expose in a spec? The specs we have proved for concurrent objects in Sections 2 and 6 allow for efficient compositional reasoning about clients, but they are also non-trivial to formulate and verify. Luckily, the FCSL way of reasoning provides a flexible solution for the compositionality-versus-complexity conundrum [31, §7].

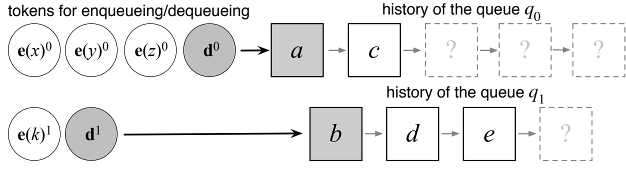


Figure 9. Tokens and histories of a balancer-based queue.

In FCSL, it is up to the library implementor to decide, how much of implementation-specific insight should go into a spec. The amount of such details is determined based on the foreseen client scenarios. For instance, we have hidden the balancer in the spec (17), but decided keep the exact constant 2, which would allow us to derive more precise quantitative bounds later (see Section 7.2). We could have hidden this component too (as well as some parts of the invariant $\mathcal{I}$), by employing in the specification sigma-types (a dependently-typed analogue of existential types), provided by FCSL as it's embedded into Coq [7]. We could have also omitted tokens from the spec, therefore, reducing the set of derivable client-specific properties to Section 6's **R1** only.

9. Mechanization and Evaluation

We have mechanized the specs and the proofs of all the examples from this paper [1], taking advantage of the fact that FCSL has been recently implemented as a tool for concurrency verification [40] on top of the Coq proof assistant [7].

Table 1 summarizes the statistics with respect to our mechanization in terms of lines of code and compilation times. The examples were proof-checked on a 3.1 GHz Intel Core i7 OS X machine with 16 Gb RAM, using Coq 8.5pl2 and Ssreflect 1.6 [17]. As the table indicates, a large fraction of the implementation is dedicated to proofs of preservation of resource invariants (Inv), *i.e.*, checking that the actual implementations do not "go wrong". In our experience, these parts of the development are the most tricky, as they require library-specific insights to define and reason about auxiliary histories. Since FCSL is a general-purpose verification framework, which does not target any specific class of programs or properties, we had to prove problem-specific facts, *e.g.*, lemmas about histories of a particular kind (Facts), and to establish the specs of interest stable (Stab). Once this infrastructure has been developed, the proofs of main procedures turned out to be relatively small (Main).

Fortunately, trickiness in libraries is invisible to clients, as FCSL proofs are compositional. Indeed, because specs are encoded as Coq types [40], the substitution principle automatically applies to programs *and proofs*. At the moment, our goal was not to optimize the proof sizes, but to demonstrate that FCSL as a tool is suitable *off-the-shelf* for machine-checked verification of properties in the spirit of novel correctness conditions [4, 22, 28]. Therefore, we didn't invest into building advanced tactics [34] for specific classes of programs [52] or properties [6, 13, 51].

Program	Facts	Inv	Stab	Main	**Total**	**Build**
Exchanger (§3)	365	1085	446	162	2058	4m 46s
Exch. Client (§5)	258	–	–	182	440	57s
Count. Netw. (§6)	379	785	688	27	1879	12m 23s
CN Client 1 (§7.1)	141	–	–	180	321	3m 11s
CN Client 2 (§7.2)	115	–	–	259	374	3m 9s

Table 1. Mechanization of the examples: lines of code for program-specific facts (Facts), resource invariants and transitions (Inv), stability proofs for desired specs (Stab), spec and proof sizes for main functions (Main), total LOC count (**Total**), and build times (**Build**). The "–" entries indicate the components that were not needed for the example.

10. Related Work

Linearizability and history-based criteria. The need for correctness criteria alternative to linearizability [25], which is more relaxed yet compositional, was recognized in the work on counting networks [4]. The suggested notion of quiescent consistency [43] required the operations separated by a quiescent state to take effect in their logical order. A more refined correctness condition, *quasi-linearizability*, implementing a relaxed version of linearizability with an upper bound on nondeterminism, was proposed by Afek *et al.* [2], allowing them to obtain the quantitative boundaries similar to what we proved in Section 7.2. The idea of relaxed linearizability was later used in the work on *quantitative relaxation* (QR) [23] for designing scalable concurrent data structures by changing the specification set of sequential histories. Most recently, *quantitative quiescent consistency* has been proposed as another criterion incorporating the possibility to reason about effects of bounded thread interference [28]. It is worth noticing that some of these correctness criteria are incomparable (*e.g.*, QC and QR [23], QL and QQC [28]) hence, for a particular concurrent object, choosing one or another criterion should be justified by the needs of the object's client. Therefore, a suitable correctness condition is essentially "*in the eye of the beholder*", as is typical in programming, when designing libraries and abstract data structures, and the logic-based approach we advocate provides precisely this flexibility in choosing desired specs.

Hoare-style specifications of concurrent objects. Hoare-style program logics were used with great success to verify a number of concurrent data structures and algorithms, which are much more natural to specify in terms of observable state modifications, rather than via call/return histories. The examples of such objects and programs include barriers [12, 26], concurrent indices [8], flat combiner [41, 47], event handlers [44], shared graph manipulations [37, 40], as well as their multiple client programs. The observation about a possibility of using program logics as a correctness criterion, alternative to linearizability, has been made in some of the prior works [8, 27, 45]. Their criticism of linearizability addressed its inability to capture the state-based

properties, such as dynamic memory ownership [27]—something that linearizability indeed cannot tackle, unless it's extended [19]. However, we are not aware of any prior attempts to capture CAL, QC and QQC-like properties of concurrent executions by means of *one and the same* program logic and employ them in client-side reasoning.

Several logics for proving linearizability or, equivalently, observational refinement [15, 49], have been proposed recently [33, 47, 50], all employing variations of the idea of using *specifications as resources*, and identifying (possibly, non-fixed or non-local) linearization points, at which such specification should be "run". In these logics, after establishing linearizability of an operation, one must still devise its Hoare-style spec, such that the spec is useful for the clients.

Similarly to the way linearizability allows one to replace a concurrent operation by an atomic one, several logics have implemented the notion of *logical atomicity*, allowing the clients of a data structure to implement application-specific synchronization on top of the data structure operations. Logical atomicity can be implemented either by parametrizing specs with client-specific auxiliary code [27, 30, 44, 45] or by engineering dedicated rules relying on the simulation between the actual implementation and the "atomic" one [9].

Instead of trying to extend the existing approaches for logical atomicity to non-linearizable objects (for which the notion of atomicity is not intuitive), we relied on a general mechanism of auxiliary state, provided by FCSL [35]. Specifically, we adopted the idea of histories as auxiliary state [41], which, however, was previously explored in the context of FCSL only for specifying linearizable structures. We introduced enhanced notation for referring directly to histories (*e.g.*, χ_{S}, χ_{O}), although FCSL's initial logical infrastructure and inference rules remained unchanged.

In this work, we do not argue that FCSL is the only logic capable of encoding custom correctness conditions and their combinations, though, we are not aware of any other work exploring a similar possibility. However, we believe that FCSL's explicit *other* subjective state component provides the most straightforward way to do so. The logics like CAP [11] and TaDA [9], from our experience and personal communication with their authors, may be capable of implementing our approach at the expense of engineering a complicated structure of capabilities to encode histories and "snapshot" interference of an environment. Other logics incorporating the generic PCM structure [29, 30, 37, 48] might be able to implement our approach, although none of these logics provide an FCSL-style rule for hiding (12) as a uniform mechanism to express explicit quiescence.

Concurrently with this work, Hemed *et al.* developed a (not yet mechanized) verification technique for CAL [22], which they applied to the exchanger and the elimination stack. Similarly to our proposal, they specify CAL-objects via Hoare logic, but using one global auxiliary history, rather than subjective auxiliary state. This tailors their system specifically to CAL (without a possibility to incorporate reasoning about other, non CA-linearizable, concurrent structures), and to programs with a *fixed* number of threads. In contrast, FCSL supports dynamic thread creation, and is capable of uniformly expressing and mechanically verifying several different criteria, with CAL merely a special case, obtained by a special choice of PCM. Moreover, in FCSL the criteria combine, as illustrated in Section 5, where we combined quiescence with CAL via hiding. Hiding is crucial for verifying clients with explicit concurrency, but is currently unsupported by Hemed *et al.*'s method.

11. Conclusion and Future Work

We have presented a number of formalization techniques, enabling specification and verification of non-linearizable concurrent objects and their clients in Hoare-style program logics. All the explored reasoning patterns involve the idea of formulating execution histories as auxiliary state, capturing the expected concurrent behavior. We have discovered that quantitative logic-based reasoning about concurrent behaviors can be done by storing relevant information about interference directly into the entries of a logical history.

We believe that our results help to bring the Hoare-style reasoning into the area of non-linearizable concurrent objects and open a number of exciting opportunities for the field of mechanized logic-based concurrency verification.

For instance, in this paper we have deliberately chosen to focus on simple client programs to showcase the specs we gave to concurrent libraries. However, any larger program incorporating these examples can be verified compositionally in FCSL, out of *these clients' specs*, via the substitution principles of FCSL [35, 40], without the need to deal with concepts such as histories and tokens that are specific to particular libraries. We believe that the reasoning patterns we have described will be useful for mechanical verification of larger weakly-synchronized approximate parallel computations [38], exploiting the QC and QQC-like behavior.

Furthermore, by ascribing interference-sensitive quantitative specs in the spirit of (17) to relaxed concurrent libraries [23], one can assess the applicability of a library implementation for its clients: the clients should tolerate the anomalies caused by interference, as long as they can logically infer the desired safety assertions from a library spec, which is fine-tuned for particular usage scenarios.

Acknowledgements We thank the anonymous reviewers from OOPSLA'16 PC and AEC for their feedback. We are grateful to Yannis Smaragdakis for his efforts as OOPSLA PC chair and to Sophia Drossopoulou for her dedication to bring out the best of the paper. This research was partially supported by the Spanish MINECO project RISCO (TIN2015-71819-P) and the US National Science Foundation (NSF). Any opinion, findings, and conclusions or recommendations expressed in the material are those of the authors and do not necessarily reflect the views of NSF.

A. Exchanger Invariants and Proof Outline

In this section, we formally define the exchanger's state invariants, and present the proof outline for its spec (10).

Additional exchanger invariants The states in the exchanger state-space must satisfy other invariants in addition to (9). These properties arise from our description of how the exchanger behaves on decorated state. We abbreviate with $p \mapsto (x; y)$ the heap $p \mapsto x \cup p{+}1 \mapsto y$.

(i) h_J contains a pointer g and a number of offers $p \mapsto (v; x)$, and g points to either null or to some offer in h_J.

(ii) χ_S, χ_O and $\|m_\mathsf{J}\|$ contain only disjoint time-stamps. Similarly, π_S is disjoint from π_O.

(iii) All offers in m_J are matched and owned by some thread: $\exists t.\, p \mapsto (t, v, w) \subseteq m_\mathsf{J} \Leftrightarrow p \in \pi_\mathsf{S} \cup \pi_\mathsf{O}, p \mapsto (v; \mathsf{M}\ w) \subseteq h_\mathsf{J}$.

(iv) There is at most one unmatched offer; it is the one linked from g. It is owned by someone: $p \mapsto (v; \mathsf{U}) \subseteq h_\mathsf{J} \implies p \in \pi_\mathsf{S} \cup \pi_\mathsf{O}, g \mapsto p \subseteq h_\mathsf{J}$. .

(v) Retired offers aren't owned: $p \mapsto (v; \mathsf{R}) \subseteq h_\mathsf{J} \Rightarrow p \notin \pi_\mathsf{S} \cup \pi_\mathsf{O}$.

(vi) The outstanding offers are included in the joint heap, *i.e.*, if $p \in \pi_\mathsf{S} \cup \pi_\mathsf{O}$ then $p \in \mathsf{dom}\ h_\mathsf{J}$.

(vii) The combined history $\chi_\mathsf{S} \cup \chi_\mathsf{O} \cup \|m_\mathsf{J}\|$ is gapless: if it contains a time-stamp t, it also contains all the smaller time-stamps (sans 0).

Explaining the proof outline Figure 10 presents the proof outline for the spec (10). We start with the precondition, and after allocation in line 2, h_S stores the offer p in line 3.

If CAS at line 4 succeeds, the program "installs" the offer; that is, the state (real and auxiliary) is changed simultaneously to the modification of g. In particular, p is added to π_S, and the offer p changes ownership, to move from h_S to h_J. Since b will be bound to null, this leads us to the assertion in line 7. We explain in Section 4 how these kinds of changes to the auxiliary state, which are supposed to occur simultaneously with some atomic operation (in this case, CAS), are specified and verified in FCSL. The assertion in line 7 further states bounded $p\ v\ \eta$. We do not formally define bounded here (it is in the proof scripts, accompanying the paper), but it says that p has been moved to h_J, *i.e.*, $p \mapsto (v; -) \subseteq h_\mathsf{J}$, and that any time-stamp t at which another thread may match p, and thus place the entry $p \mapsto (t, v, -)$ into m_J, must satisfy $\mathsf{last}(\eta) < t, \bar{t}$. Intuitively, this property is valid, and stable under interference, because entries in m_J can be added only by generating fresh time-stamps wrt. the collective history $\chi_\mathsf{O} \cup \|m_\mathsf{J}\|$, and η is a subset of it. If CAS in line 4 fails, then nothing changes, so we move to the spec in line 15.

At line 8, CAS succeeds if $x = \mathsf{U}$, and fails if $x = \mathsf{M}\ w$. Notice that x cannot be R; since we own $p \in \pi_\mathsf{S}$, no other thread could retire p. If CAS fails, then the offer has been matched with w. CAS simultaneously "collects" the offer as follows. By invariant (iii), and bounded $p\ v\ \eta$, the auxiliary map m_J contains an entry $p \mapsto (t, v, w)$, where $\mathsf{last}(\eta) < t, \bar{t}$. The auxiliary state is changed to remove p from m_J, and simultaneously place $t \mapsto (v, w)$ into χ_S. If CAS succeeds, the offer was unmatched, and is "retired" by removing p from π_S. Lines 12-13 branch on x, selecting either the assertion 10 or 11, so the postcondition follows.

After reading cur in line 18, by invariant (i), we know that cur either points to null, or to some offer $p \mapsto (w; -) \subseteq h_\mathsf{J}$.

At line 24, the CAS succeeds if $x = \mathsf{U}$ and fails otherwise. If CAS succeeded, then it "matches" the offer in cur; that is, it writes $\mathsf{M}\ w$ into the hole of cur, and changes the auxiliary state as follows. It takes t to be the smallest unused time-stamp in the history $\chi = \chi_\mathsf{S} \cup \chi_\mathsf{O} \cup \|m_\mathsf{J}\|$. Thus $\mathsf{last}(\chi) < t$, and because χ has even size by invariant (9), t must be odd, and hence $t < \bar{t} = t + 1$. The $t \mapsto (v, w)$ is placed into χ_S, giving us assertion 26. To preserve the invariant (iii), CAS simultaneously puts the entry $p \mapsto (t, w, v)$ into m_J, for future collection by the thread that introduced offer cur. But, we do not need to reflect this in line 26. If the CAS fails, the history χ_S remains empty, as no matching is done. However, the hole y associated with cur cannot be U, as then CAS would have succeded. Therefore, it is sound in line 28 to "unlink" cur from g, as the unlinking will not violate the invariant (iv), which says that an unmatched offer must be pointed to by g. Finally, lines 30 and 33 select the assertion 26 or 27, and either way, directly imply the postcondition.

1 $\{h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \emptyset, \chi_\mathsf{S} = \emptyset, \eta \subseteq \chi_\mathsf{O} \cup \|m_\mathsf{J}\|\}$
2 $\quad p \leftarrow \mathtt{alloc}\ (v, \mathsf{U})$;
3 $\{h_\mathsf{S} = p \mapsto (v; \mathsf{U}), \pi_\mathsf{S} = \emptyset, \chi_\mathsf{S} = \emptyset, \eta \subseteq \chi_\mathsf{O} \cup \|m_\mathsf{J}\|\}$
4 $\quad b \leftarrow \mathtt{CAS}\ (g, \mathtt{null}, p)$;
5 $\quad$ **if** $b == \mathtt{null}$ **then**
6 $\quad\quad \mathtt{sleep}\ (50)$;
7 $\{h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \{p\}, \chi_\mathsf{S} = \emptyset, \eta \subseteq \chi_\mathsf{O} \cup \|m_\mathsf{J}\|, \mathtt{bounded}\ p\ v\ \eta\}$
8 $\quad\quad x \leftarrow \mathtt{CAS}\ (p{+}1, \mathsf{U}, \mathsf{R})$;
9 $\{h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \emptyset, \eta \subseteq \chi_\mathsf{O} \cup \|m_\mathsf{J}\|,$
10 $\; x = \mathsf{M}\ w \implies \exists t.\, \chi_\mathsf{S} = t \mapsto (v, w), \mathsf{last}(\eta) < t, \bar{t},$
11 $\; x = \mathsf{U} \implies \chi_\mathsf{S} = \emptyset\}$
12 $\quad\quad$ **if** x **is** $\mathsf{M}\ w$ **then return** $(\mathtt{Some}\ w)$
13 $\quad\quad$ **else return** $\mathtt{None}$
14 $\quad$ **else**
15 $\{h_\mathsf{S} = p \mapsto (v; \mathsf{U}), \pi_\mathsf{S} = \emptyset, \chi_\mathsf{S} = \emptyset, \eta \subseteq \chi_\mathsf{O} \cup \|m_\mathsf{J}\|\}$
16 $\quad\quad \mathtt{dealloc}\ p$;
17 $\{h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \emptyset, \chi_\mathsf{S} = \emptyset, \eta \subseteq \chi_\mathsf{O} \cup \|m_\mathsf{J}\|\}$
18 $\quad\quad cur \leftarrow \mathtt{read}\ g$;
19 $\{h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \emptyset, \chi_\mathsf{S} = \emptyset, \eta \subseteq \chi_\mathsf{O} \cup \|m_\mathsf{J}\|,$
20 $\; cur = \mathtt{null} \vee cur \mapsto (w; -) \subseteq h_\mathsf{J}\}$
21 $\quad\quad$ **if** $cur == \mathtt{null}$ **then return** $\mathtt{None}$
22 $\quad\quad$ **else**
23 $\{h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \emptyset, \chi_\mathsf{S} = \emptyset, \eta \subseteq \chi_\mathsf{O} \cup \|m_\mathsf{J}\|, cur \mapsto (w; -) \subseteq h_\mathsf{J}\}$
24 $\quad\quad\quad x \leftarrow \mathtt{CAS}(cur{+}1, \mathsf{U}, \mathsf{M}\ v)$;
25 $\{h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \emptyset, \eta \subseteq \chi_\mathsf{O} \cup \|m_\mathsf{J}\|, cur \mapsto (w; y) \subseteq h_\mathsf{J},$
26 $\; x = \mathsf{U} \implies y = \mathsf{M}\ v, \exists t.\, \chi_\mathsf{S} = t \mapsto (v, w), \mathsf{last}(\eta) < t, \bar{t},$
27 $\; x \neq \mathsf{U} \implies \chi_\mathsf{S} = \emptyset, y \neq \mathsf{U}\}$
28 $\quad\quad\quad \mathtt{CAS}\ (g, cur, \mathtt{null})$;
29 {same as above; the state satisfies (iv) because $y \neq \mathsf{U}$}
30 $\quad\quad\quad$ **if** $x == \mathsf{U}$ **then** $w \leftarrow \mathtt{read}\ cur$; **return** $(\mathtt{Some}\ w)$
31 $\{h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \emptyset, \eta \subseteq \chi_\mathsf{O} \cup \|m_\mathsf{J}\|, \mathsf{res} = \mathsf{Some}\ w,$
32 $\; \exists t. \chi_\mathsf{S} = t \mapsto (w, v), \mathsf{last}(\eta) < t, \bar{t}\}$
33 $\quad\quad\quad$ **else return** $\mathtt{None}\}$
34 $\{h_\mathsf{S} = \emptyset, \pi_\mathsf{S} = \emptyset, \chi_\mathsf{S} = \emptyset, \eta \subseteq \chi_\mathsf{O} \cup \|m_\mathsf{J}\|, \mathsf{res} = \mathsf{None}\}$

Figure 10. Proof outline for the exchanger.

References

[1] FCSL: Fine-grained Concurrent Separation Logic. Coq Development and Code Commentary. Available on the project website at `http://software.imdea.org/fcsl`.

[2] Y. Afek, G. Korland, and E. Yanovsky. Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. In *OPODIS*, pages 395–410. Springer, 2010.

[3] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *STOC*, pages 1–9. ACM, 1983.

[4] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *J. ACM*, 41(5):1020–1048, 1994.

[5] C. J. Bell, A. W. Appel, and D. Walker. Concurrent separation logic for pipelined parallelization. In *SAS*, pages 151–166. Springer, 2010.

[6] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Tractable refinement checking for concurrent objects. In *POPL*, pages 651–662. ACM, 2015.

[7] Coq Development Team. *The Coq Proof Assistant Reference Manual - Version 8.5pl2*, 2016. `https://coq.inria.fr`.

[8] P. da Rocha Pinto, T. Dinsdale-Young, M. Dodds, P. Gardner, and M. J. Wheelhouse. A simple abstraction for complex concurrent indexes. In *OOPSLA*, pages 845–864. ACM, 2011.

[9] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A Logic for Time and Data Abstraction. In *ECOOP*, pages 207–231. Springer, 2014.

[10] J. Derrick, B. Dongol, G. Schellhorn, B. Tofan, O. Travkin, and H. Wehrheim. Quiescent Consistency: Defining and Verifying Relaxed Linearizability. In *FM*, pages 200–214. Springer, 2014.

[11] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *ECOOP*, pages 504–528. Springer, 2010.

[12] M. Dodds, S. Jagannathan, and M. J. Parkinson. Modular reasoning for deterministic parallelism. In *POPL*, pages 259–270. ACM, 2011.

[13] C. Dragoi, A. Gupta, and T. A. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In *CAV*, pages 174–190. Springer, 2013.

[14] Class `Exchanger<V>`, Java Platform SE 8 Documentation. Available from `http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Exchanger.html`. Accessed June 24, 2015.

[15] I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

[16] M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*, pages 388–402. Springer, 2010.

[17] G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Technical Report 6455, Microsoft Research – Inria Joint Centre, 2009.

[18] A. Gotsman, N. Rinetzky, and H. Yang. Verifying concurrent memory reclamation algorithms with grace. In *ESOP*, pages 249–269. Springer, 2013.

[19] A. Gotsman and H. Yang. Linearizability with ownership transfer. In *CONCUR*, pages 256–271. Springer, 2012.

[20] A. Haas, T. A. Henzinger, A. Holzer, C. M. Kirsch, M. Lippautz, H. Payer, A. Sezgin, A. Sokolova, and H. Veith. Local Linearizability for Concurrent Container-Type Data Structures. In *CONCUR*, 2016. To appear.

[21] N. Hemed and N. Rinetzky. *Brief announcement*: Concurrency-Aware Linearizability. In *PODC*, pages 209–211. ACM, 2014.

[22] N. Hemed, N. Rinetzky, and V. Vafeiadis. Modular verification of concurrency-aware linearizability. In *DISC*, pages 371–387. Springer, 2015.

[23] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. In *POPL*, pages 317–328. ACM, 2013.

[24] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. M. Kaufmann, 2008.

[25] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, 1990.

[26] A. Hobor and C. Gherghina. Barriers in concurrent separation logic. In *ESOP*, pages 276–296. Springer, 2011.

[27] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282. ACM, 2011.

[28] R. Jagadeesan and J. Riely. Between Linearizability and Quiescent Consistency - Quantitative Quiescent Consistency. In *ICALP (2)*, pages 220–231. Springer, 2014.

[29] R. Jung, R. Krebbers, L. Birkedal, and D. Dreyer. Higher-order ghost state. In *ICFP*. ACM, 2016.

[30] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650. ACM, 2015.

[31] L. Lamport. Composition: A way to make proofs harder. In *COMPOS*, pages 402–423. Springer, 1998.

[32] R. Ley-Wild and A. Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, pages 561–574. ACM, 2013.

[33] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470. ACM, 2013.

[34] A. McCreight. Practical Tactics for Separation Logic. In *TPHOLs*, pages 343–358. Springer, 2009.

[35] A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, pages 290–310. Springer, 2014.

[36] S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[37] A. Raad, J. Villard, and P. Gardner. CoLoSL: Concurrent Local Subjective Logic. In *ESOP*, pages 710–735. Springer, 2015.

[38] M. C. Rinard. Unsynchronized techniques for approximate parallel computing. In *RACES - SPLASH Workshop*, 2012.

[39] W. N. Scherer III, D. Lea, and M. L. Scott. A scalable elimination-based exchange channel. In *SCOOL*, 2005.

[40] I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87. ACM, 2015.

[41] I. Sergey, A. Nanevski, and A. Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *ESOP*, pages 333–358. Springer, 2015.

[42] N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.

[43] N. Shavit and A. Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, 1996.

[44] K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. In *ESOP*, pages 149–168. Springer, 2014.

[45] K. Svendsen, L. Birkedal, and M. J. Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP*, pages 169–188. Springer, 2013.

[46] R. K. Treiber. Systems programming: coping with parallelism. Technical Report RJ 5118, IBM Almaden, 1986.

[47] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390. ACM, 2013.

[48] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, pages 691–707. ACM, 2014.

[49] A. J. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *POPL*, pages 343–356. ACM, 2013.

[50] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.

[51] V. Vafeiadis. Automatically proving linearizability. In *CAV*, pages 450–464. Springer, 2010.

[52] K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361. ACM, 2008.