A Complete Program Logic for Compositional Linearizability

Eashan Hatti
Yale University, New Haven, CT, USA

Arthur Oliveira Vale
Yale University, New Haven, CT, USA

Zhongye Wang
Yale University, New Haven, CT, USA

Yueyang Feng
Yale University, New Haven, CT, USA

Zhong Shao
Yale University, New Haven, CT, USA

Abstract

We present Linearizability Hoare Logic (LHL), the first *mechanized*, *sound*, and *complete* program logic for atomic, set, and interval linearizability. We achieve this by showing soundness and completeness of LHL w.r.t. a more general criterion, *compositional linearizability*, which subsumes all three criteria. We showcase the expressivity of LHL by verifying an exchanger with a set linearizable specification, the elimination-backoff stack built above the exchanger, a lock with an atomic linearized specification, and a write-snapshot object with an interval linearizable specification.

Together with LHL we formalize a modular verification framework for concurrent components based on the theory of compositional linearizability. This allows us to specify components at a high level of abstraction and granularity, and then assemble them into large systems that are correct by construction. As a showcase, we verify the elimination-backoff stack modularly by verifying each of its sub-components against their linearized specifications and then linking them together.

2012 ACM Subject Classification Theory of computation → Program verification; Computing methodologies → Concurrent computing methodologies; Theory of computation → Programming logic

Keywords and phrases Program Logic, Rely-Guarantee, Linearizability, Compositional Verification, Concurrency

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2026.3

Related Version *Extended Version*: `https://flint.cs.yale.edu/publications/lhl.html` [13]

Supplementary Material *Software*: `https://github.com/ehatti/LHL/tree/ecoop-camera-ready`

Funding This work is supported in part by NSF grants 2019285, 2313433, 2442888, and by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590130. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

Acknowledgements We would like to thank the anonymous reviewers and artifact evaluation committee for their thoughtful feedback and time spent reviewing our work.

© Eashan Hatti, Arthur Oliveira Vale, Zhongye Wang and Zhong Shao;
licensed under Creative Commons License CC-BY 4.0
40th European Conference on Object-Oriented Programming (ECOOP 2026).
Editors: Robbert Krebbers and Alexandra Silva; Article No. 3; pp. 3:1–3:45

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

1.1 Compositionality.

When designing large computer systems, it is good practice to encapsulate independent functionalities in separate components. This benefits verification, as a well-organized system decomposes into small open components that are completely characterized by their own implementation and their interfaces with the rest of the system.

In this context, given a module M that implements some object with interface F by importing a library with interface E, we call the implemented object the *overlay* of the module and the imported library its *underlay*. To verify the module against its overlay specification, we only consider the relevant underlay specifications, allowing provers to write independent proofs for separate modules. This means that the internals of the underlay module do not matter for other parts of the system: the module is characterized by its underlay and overlay specifications. This mode of verification is only possible in a *compositional* verification framework, where each module can be given a correctness property independent of the rest of the system, and then can be used as a building block to construct a larger system, with no side-conditions for the correctness of their compositions.

In the context of concurrent objects, linearizability [15] has been the gold standard since the 90s. It allows programmers to abstract the behavior of a concurrent object E's code `M_E` into a simpler specification `V_E` that is atomic[1] and easier to reason about. While there are many efforts to enable the mechanized verification of linearizability, few are truly scalable: either there are concurrent objects like the exchanger with non-atomic specifications that cannot be expressed in these pre-existing frameworks due to their over-reliance on atomicity, or the specification cannot be used as a black box encapsulating its code to modularly verify another object that uses this underlay object.

1.2 Compositional Linearizability.

Over time, linearizability, which was an inherently atomic correctness criterion, has been extended to handle non-atomic objects [24, 5]. Recently, [26, 25] have also presented an analysis of compositional models of computation which reveals that linearizability, even when fully generalized to handle non-atomic blocking concurrent objects, has an underlying compositional structure, which can be exploited to develop a compositional refinement calculus for the verification of concurrent modules. As an application of their theory, [26] develop an axiomatic technique for showing linearizability of individual traces, and then package it as a rely-guarantee program logic. They call their linearizability criterion and underlying framework *compositional linearizability*.

While an important first step towards the compositional verification of non-atomic concurrent objects, and a convincing showcase of the compositional linearizability theory, the verification technique proposed by [26] could still be improved for the sake of practicality in large systems verification. Some immediate issues are that their framework has not been validated by a mechanization, and while [26] give a paper proof of soundness, they only conjecture its completeness. In this paper, we set out to mechanize the verification technique and formalize the soundness proof of [26], while also proving its completeness. Completeness

[1] By *atomic* we mean that when a thread makes an invocation it receives its response immediately after, with no interference from other threads.

is the main contribution of our work, and results in the first complete program logic for any of the standard generalizations of linearizability beyond atomicity.

In doing so, we identified an additional issue for the sake of practicality. To exploit the trace-based theory they had developed earlier, [26] use traces as a universal notion of state. Unfortunately, in practice, using traces as state leads to unnecessary complexity in defining predicates, as simple properties that would be readily available in a concrete representation of state become properties that must be stated inductively over the underlying trace.

So, in order to enable us to verify complex examples, we found it paramount to switch to a state-based representation of proof configurations. It turns out that switching from trace-based reasoning to state-based reasoning is not a simple task. There are two aspects to this. First, [26] use a technique inspired by [15] which they call possibility reasoning, where one attempts to maintain a set of *possible* linearizations for the current computation through a representation of linearizations in a structure they call a *possibility*. Unfortunately, some of the possibility update rules of [26] do not readily generalize to a state-based formulation, as they involve manipulating the ordering of events on a trace.

Another aspect of switching to state-based reasoning is that many of the algebraic properties [26] rely on to show the compositionality of the verification framework no longer hold strictly. Technically, their trace-based representation allows them to work up to equality to show several categorical properties of their refinement calculus. We had to generalize these categorical properties to hold only up to notions of equivalence in order to generalize their results to concrete states. In the process, we end up mechanizing a large portion of a generalization of the theory in [26].

Summary and Main Contributions

- We present the first *mechanized* program logic, Linearizability Hoare Logic (LHL), which supports and unifies the verification of Herlihy-Wing linearizability and other non-atomic linearizabilities into a generalized criterion called *compositional linearizability* [26].
- To mechanize compositional linearizability, we first formalize a compositional model of computation in §3, formalizing modules, specifications, objects, refinement and composition operations, and proving their algebraic and compositional properties.
- In §4, we generalize compositional linearizability, originally based solely on traces, to our LTS setting, along with its compositional properties.
- Finally, we present proof rules of our LHL in §5, which uses a technique called *possibility reasoning* to construct linearizability proofs. We prove its soundness and completeness w.r.t. compositional linearizability.
- Using LHL, we prove several representative examples, including: a modular proof of the elimination-backoff stack [14] in the structure proposed in Fig. 1, to be explained throughout the paper; we mechanize a lock and then a coarse-grained locked racy object (the motivating example used by [26]); to showcase full-blown non-atomicity, we also mechanize an interval-linearizable [5] write-snapshot object.

All results of the paper are mechanized in Rocq and are found as supplemental material. In addition, an extended version [13] of this paper is available containing further details on the elimination-backoff stack and a detailed comparison between our paper and [26].

2 Motivating Example

Throughout the paper, we use the *elimination-backoff stack* [14] (henceforth “EBStack”) to demonstrate our approach.

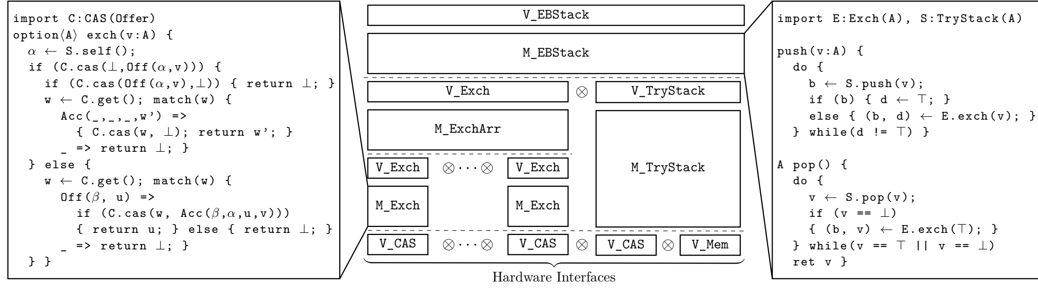


Figure 1 The exchanger implementation (left), and the hierarchy (middle) of underlay object implementations (`M_xxx`) and their interfaces (`V_xxx`) used in the elimination-backoff stack implementation (right): A is the type of values stored in the stack; `Offer` is a data type defined by `Offer` $:=$ `Off`(α, v) | `Acc`(α, β, v, w), where $\alpha, \beta \in \mathcal{T}$ are thread identifiers and $v, w \in A$; The $\bot$ value indicates the failure of the operation in both the exchanger and the EBStack object.

The EBStack can be decomposed into multiple concurrent objects as shown in Fig. 1. Each object implements a specific overlay specification defined as a labelled transition system (LTS), and only uses operations provided in its underlay interfaces. For example, the top-level EBStack object implements the usual stack specification `V_EBStack` above the exchanger object and a try stack (a stack whose operations may fail). As shown in the code in the right box, the EBStack will first try to perform a push/pop operation through the try stack. If the operation fails, the EBStack tries to eliminate with another operation through the `exch` operation of the exchanger object, which only succeeds if there are concurrent `exch` operations. The EBStack's push/pop operations loop until one of these two steps succeeds. The exchanger implementation is encapsulated in a different module `Exch`, which is built above the CAS memory cells. To increase throughput, an array of exchangers is used instead of the single exchanger in the EBStack [14]. In § 3, we explain in detail the programming language we use to implement these objects and LTS for specifying objects.

The key benefit of this layered approach is that, with proper encapsulation, the implementation and verification of each object become relatively concise, manageable, and, most importantly, independent of other objects. In the hierarchy in Fig. 1, the `EBStack` code depends only on the `V_Exch` and `V_TryStack` interfaces and is decoupled from the implementation and verification of these overlay objects. After verifying the `EBStack` layer, its proof will remain untouched when the implementation of its underlay objects is modified, as they are all verified to obey the object specifications that are independent of the implementation code. For example, the exchanger array `ExchArr` and the single exchanger `Exch` share the same overlay specification `V_Exch` because they are meant to implement the same operation with the same functionality, and, as a result, we may use either one as the `V_Exch` library in the overlay `EBStack` object without any modification to any implementation code or proof.

This leads to the reason that we are interested in the verification of the elimination-backoff stack. The variations of EBStack shown in Fig. 1 have been verified in other concurrent object verification frameworks [22, 11]. However, none of them achieves the same degree of compositionality shown in this diagram. They all choose to inline the code of the exchanger into the code of the EBStack, which results in complicated EBStack code and proof. Some choose to inline the single exchanger code, which brings another problem: when they want to replace the single exchanger with a more performant implementation like the exchanger array, they need to redo the entire proof of the EBStack. We further discuss the comparison between their proofs and ours in § 6.

The reason for this is that their specifications and correctness criteria cannot support non-atomic objects, such as the exchanger object. An exchange only makes sense if two threads are executing concurrently, which is not possible in an atomic specification. Therefore, the exchanger object does not admit a deterministic atomic linearized specification. Hence, it cannot be verified as a separate module with strict proof encapsulation, and its proof has to be merged into the proof of the overlay stack object. We use the theory of compositional linearizability to specify the exchanger by a set-linearizable LTS, and use a program logic based on possibility reasoning to verify the exchanger and other objects as standalone modules. Compositionality ensures the correctness of the top-level stack w.r.t. its specification `V_EBStack` in a system running above the hardware interfaces, given the per-module correctness w.r.t. their own overlay specs. We elaborate on this verification technique in § 4 and § 5.

3 Programming Language and Specifications

Our framework verifies concurrent *modules* M : `Mod E F`, which implement operations specified by an *effect signature* `F`. A module M implements each operation $f \in$ `F` as a program M^f : `Prog E` $\mathtt{ar}(f)$, which uses operations from an underlay signature `E` and returns a value in the return type (specified by $\mathtt{ar}(f)$) of the operation `f` that it implements. An object over an effect signature `E` is specified by a state-transition system `V_E : Spec E`. The core verification task is to show that given a specification `V_E : Spec E`, called the *underlay* of M, the module M : `Mod E F` correctly implements its overlay specification `V_F : Spec F`. We now discuss these notions formally.

3.1 Effect Signatures

An effect signature consists of a set `E` of effects e and two assignments $\mathtt{par}_{\mathtt{E}}(-) : \mathtt{E} \to \mathtt{Set}$ and $\mathtt{ar}_{\mathtt{E}}(-) : \mathtt{E} \to \mathtt{Set}$, specifying respectively a set of *parameters* and a set of *return* values for each effect $e \in \mathtt{E}$. We find it convenient to encapsulate this information in the following notation, whose usage will become clear shortly:

$$\mathtt{E} := \{e : \mathtt{par}_{\mathtt{E}}(e) \to \mathtt{ar}_{\mathtt{E}}(e) \mid e \in \mathtt{E}\}$$

When it causes no confusion, we omit the subscript on $\mathtt{par}_{\mathtt{E}}(-)$ and $\mathtt{ar}_{\mathtt{E}}(-)$.

In the context of our paper, effects $e \in E$ are method names, which take as arguments elements of the parameter set $\mathtt{par}(e)$ and return some value in its arity $\mathtt{ar}(e)$. For instance, the signature for the EBStack data structure is the following, where A is the type of stack elements and $\mathbf{1}$ is the unit type.

$$\mathtt{EBStack} := \{\mathtt{push} : A \to \mathbf{1}, \mathtt{pop} : \mathbf{1} \to A + \mathbf{1}\}$$

Signatures also support a *union* operation, which collects the operations of both signatures into a single, combined signature. Formally:

$$\mathtt{E} + \mathtt{F} := \{\mathtt{inl}\ \ e : \mathtt{par}_{\mathtt{E}}(e) \to \mathtt{ar}_{\mathtt{E}}(e) \mid e \in \mathtt{E}\} \uplus \{\mathtt{inr}\ \ e : \mathtt{par}_{\mathtt{F}}(e) \to \mathtt{ar}_{\mathtt{F}}(e) \mid e \in \mathtt{F}\}$$

3.2 Programs

Programs play the role of method bodies within modules in our mechanization. We formalize programs as *interaction trees* [33], which we type as `Prog`. Interaction trees are coinductively defined data structures with the following syntax:

$$p \in \mathtt{Prog\ E}\, R := \mathtt{Vis}\, f(a)\, k \mid \mathtt{Ret}\, r \mid \mathtt{Tau}\, p \text{ where } f \in \mathtt{E} \quad a \in \mathtt{par}(f) \quad k : \mathtt{ar}(f) \to \mathtt{Prog\ E}\, R \quad r \in R$$

194 A program $p \in$ `Prog E` R represents a (possibly infinite) series of operations that result in a
195 return value of type R (if finite). The three constructors function as follows.
196 **`Vis` f(a) k:** Issues an invocation to the operation $f(a)$ of the underlay signature `E`, and
197 proceeds with the continuation $k(v)$ after receiving a return value $v \in \mathtt{ar}(f)$.
198 **`Ret` r:** Once a program executes all of its operations, it terminates by returning a value r of
199 type R.
200 **`Tau` p:** Represents a silent step and is necessary for looping constructs to satisfy Rocq's
201 productivity checker. We refer readers to [33] for more details.

202 `Prog E _` defines a monad for any `E`, which allows for programs to be written in a monadic
203 style. The program bodies in Fig. 1 are the sugared version of a monadic expression over
204 `Prog E _`. The additional control structures, such as the `if` conditional and the `while` loop
205 constructs used in Fig. 1, may be defined in a standard way.

206 3.3 Modules

207 A module M : `Mod E F` implements operations of a signature `F`, using the operations of
208 the signature `E`. A module M : `Mod E F` consists of a mapping from operations $f \in$ `F` and
209 arguments $a \in \mathtt{par}(f)$ to programs $M^{f(a)}$: `Prog E` $\mathtt{ar}(f)$, implementing a call $f(a)$ as a
210 program operating over `E` and returning a value in its return type $\mathtt{ar}(f)$. In other words,
211 a module M : `Mod E F` is a collection $M : \Pi_{f \in F}(\mathtt{par}(f) \to \mathtt{Prog}\ E\ \mathtt{ar}(f))$. For instance,
212 $M_{\mathtt{EBStack}}$ is the mapping taking `push(v)` to the program described in Fig. 1 and similarly
213 for `pop()`. Nonetheless, we write $M^{f(a)} : \mathtt{Prog}\ E\ \mathtt{ar}(f)$ for the program corresponding to
214 $f \in F$ and $a \in \mathtt{par}(f)$ in M, and $M^f : \mathtt{par}(f) \to \mathtt{Prog}\ E\ \mathtt{ar}(f)$.
215 Modules may be composed vertically by interaction tree substitution. Given modules
216 M : `Mod E F` and N : `Mod F G`, their vertical composition $M :\triangleright N$: `Mod E G` implements the
217 operation $g(a)$ of `G` by running the corresponding program $N^{g(a)}$ with method invocations
218 $f(a')$ to `F` replaced by their corresponding programs $M^{f(a')}$. We refer readers to the
219 mechanization for the exact definition. Crucially, the vertical composition operation admits
220 an identity element:

221 $$\mathtt{idM}_E : \mathtt{Mod\ E\ E} := (\mathtt{idProg}_e)_{e \in \mathtt{E}} \quad \text{where } \mathtt{idProg}^e := \lambda(a : \mathtt{par}(e)).\mathtt{Vis}\ e(a)\ \ \mathtt{Ret}$$

222 The vertical composition operation ($:\triangleright$) and its identity element satisfy the following
223 properties:

224 $$(1)\ \mathtt{idM} :\triangleright M \approx M \qquad (2)\ M :\triangleright \mathtt{idM} \approx M \qquad (3)\ (M_1 :\triangleright M_2) :\triangleright M_3 \approx M_1 :\triangleright (M_2 :\triangleright M_3)$$

225 where $\approx$ is an equivalence relation defined on modules as the pointwise equivalence of
226 programs up to removal of silent steps (`Tau`). The first two properties show that `idM` is the
227 neutral element for vertical composition, up to $\approx$ equivalence, and the third property is its
228 associativity.

229 3.4 Specifications

230 We now give a formal definition of *labelled transition system* (LTS), the *specifications* we use
231 for objects. Recall that a module M : `Mod E F` depends on the operations provided by `E` to
232 implement the operations of `F`. To define the operational semantics of a module, we first
233 define the behavior of the operations in `E` using an LTS.
234 First, we must define what kinds of events we are interested in. We assume as given a set
235 of thread names $\mathcal{T}$, which is a parameter over the whole model. We use a set of the form

$\{i \in \mathbb{N} \mid i < T\}$, where $T \in \mathbb{N}$ is the total number of threads, as $\mathcal{T}$. In a multicore setting, $\mathcal{T}$ is the set of cores available; in a multi-threaded setting, it is the set of available threads. While users can fix the whole set $\mathcal{T}$ when verifying an object, we choose to parameterize our theories and proofs over $\mathcal{T}$. As a result, our proofs of linearizability universally quantify over such finite sets of thread names.

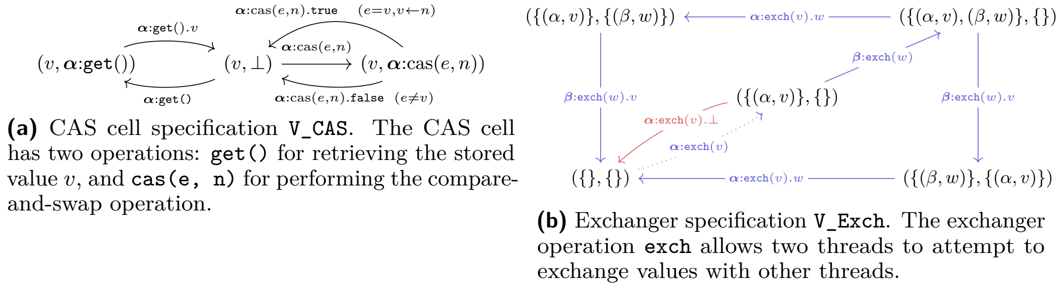
Then, an event of the signature E consists of

$$\texttt{TEvent E} := \boldsymbol{\alpha}{:}f(a) \mid \boldsymbol{\alpha}{:}f(a).v$$

where $\alpha \in \mathcal{T}$, $f \in \texttt{E}$, $a \in \texttt{par}(f)$ and $v \in \texttt{ar}(f)$. An event is either $\boldsymbol{\alpha}{:}f(a)$, a call to an operation of E, or $\boldsymbol{\alpha}{:}f(a).v$, a return event tagged with the thread that issued it. Then, a labelled transition system V over E, written $\texttt{V} : \texttt{Spec E}$, is defined as a triple $(S, \rightarrow_{\texttt{V}}, s_0)$ of a set of states S, a transition relation $\rightarrow_{\texttt{V}} \subseteq S \times \texttt{TEvent E} \times S$, and an initial state $s_0 \in S$.

Unlike many approaches for modeling concurrent specifications, which assume operations are logically atomic, the call and return events are separate transitions in our specifications. This is necessary to model objects that are not linearizable to atomic specifications. We do, however, require that the labelled transition system be *locally sequential*, i.e., the projection to each thread is sequential (it strictly alternates between invocations and responses). As a result, we must carry some extra state, like the operation that is currently executing (e.g., Fig. 2a), so we prevent other calls from happening until the current pending call is completed.

We now give a couple of examples of linearized state transition systems used in the verification of the EBStack. In our state diagrams, such as Fig. 2a, we use conditionals and state updates in the labels, and depict the transitions for abstracted thread names α, β, to mean that for any thread α a certain transition exists (if threads have different names they are distinct $\alpha \neq \beta$). This allows us to make the diagrams finite so they may be depicted succinctly on paper.



(a) CAS cell specification `V_CAS`. The CAS cell has two operations: `get()` for retrieving the stored value v, and `cas(e, n)` for performing the compare-and-swap operation.

(b) Exchanger specification `V_Exch`. The exchanger operation `exch` allows two threads to attempt to exchange values with other threads.

Figure 2 CAS and Exchanger Linearized Specifications

▶ **Example 1** (Atomic CAS Cell LTS). We begin with an atomic specification encoded by the LTS in Fig. 2a for a CAS cell object. Its states are tuples consisting of the stored value and the ongoing pending call. A non-empty pending call component restricts the next transition to be labelled with the matching return event. The two transitions on the left ensure that a get operation $\boldsymbol{\alpha}{:}\texttt{get}()$ always gets the stored value v in the return event $\boldsymbol{\alpha}{:}\texttt{get}().v$. The three transitions on the right specify that a CAS operation $\boldsymbol{\alpha}{:}\texttt{cas}(e, n)$ receives `true` as a response only when the stored value is e and stores the new value n to it, and otherwise, it receives `false` and does not modify the stored value. This LTS specifies an atomic object because, after an invocation by thread α, the next transition must be a response by the same thread α. Invocation transitions are possible only when the ongoing operation is $\bot$, resulting in only sequential executions.

271 ▶ **Example 2** (Set-Linearizable Exchanger LTS). Fig. 2b defines the LTS specification for the
272 exchanger object. Its LTS state consists of two sets over tuples of a thread name and an
273 exchanged value. The left and right sets are the "offer set" and "accept set", containing
274 respectively the exchange offers that threads have proposed and accepted. In a successful
275 exchange, two threads will add their offers to the offer set, after which one thread accepts the
276 other offer. The remaining thread will then accept another offer and empty the set. Tracing
277 through the *blue* arrows, we can extract the execution of this successful exchange as trace
278 (1).

279 $$\boldsymbol{\alpha}{:}\texttt{exch}(v) \cdot \boldsymbol{\beta}{:}\texttt{exch}(w) \cdot \boldsymbol{\alpha}{:}\texttt{exch}(v).w \cdot \boldsymbol{\beta}{:}\texttt{exch}(w).v \tag{1}$$

280 In a failed exchange, the offering thread will revoke its offer and return $\bot$. This will step
281 through the dotted blue arrow and the *red* arrow as trace (2).

282 $$\boldsymbol{\alpha}{:}\texttt{exch}(v) \cdot \boldsymbol{\alpha}{:}\texttt{exch}(v).\bot \tag{2}$$

283 All possible executions of the exchanger in the specification are repetitions of trace (1) and
284 trace (2). In trace (1), calls are not necessarily followed immediately by their returns – the
285 calls and returns of both threads α and β are interleaved with each other. As a result, this
286 specification is **not** atomic linearizable but is *set linearizable* [24]: calls and returns in the
287 exchanger's execution traces always come as a set of at most two calls followed by a set of
288 matching returns.

289 3.5 Traces and Refinement

We define a trace over a signature $\texttt{E}$ as a list $t \in (\texttt{TEvent E})^*$ of thread events. We write $\texttt{Trace E}$ for the set of traces over $\texttt{E}$. The LTS specification $\texttt{V} : \texttt{Spec E}$ specifies the behavior of an object through $[\![\texttt{V}]\!]$, a set of traces defined as follows: we first define a path in $\texttt{V} = (S, \rightarrow_{\texttt{V}}, s_0)$ and then take all paths starting from the initial state as the set of behaviors.

$$\frac{s \in S}{s \xrightarrow{\epsilon}_{\texttt{V}} s} \qquad \frac{s_1 \rightarrow^{\texttt{ev}}_{\texttt{V}} s_2}{s_1 \xrightarrow{\texttt{ev}}_{\texttt{V}} s_2} \qquad \frac{s_1 \xrightarrow{t}_{\texttt{V}} s_2 \qquad s_2 \xrightarrow{t'}_{\texttt{V}} s_3}{s_1 \xrightarrow{t \cdot t'}_{\texttt{V}} s_3} \qquad [\![\texttt{V}]\!] := \{t \in (\texttt{TEvent E})^* \mid \exists s \in S. s_0 \xrightarrow{t}_{\texttt{V}} s\}$$

Our notion of refinement is then just behavior containment:

$$\texttt{V} \sqsupseteq \texttt{V}' \iff [\![\texttt{V}]\!] \subseteq [\![\texttt{V}']\!] \qquad\qquad \texttt{V} \equiv \texttt{V}' \iff \texttt{V} \sqsupseteq \texttt{V}' \land \texttt{V}' \sqsupseteq \texttt{V}$$

290 It is straightforward to check that refinement is reflexive and transitive, making it a preorder.

291 3.6 Operational Semantics

292 Given an underlay specification $\texttt{V_E} : \texttt{Spec E}$, where $\texttt{V_E} = (S, \rightarrow_{\texttt{V}}, s_0)$, and a module
293 $M : \texttt{Mod E F}$, we define the behaviors of M running on top of $\texttt{V_E}$ operationally as a concrete
294 LTS ($\texttt{V_E} \triangleright M$). In §4, we define the linearizable relation of the concrete LTS w.r.t. the
295 specification LTS (those in Fig. 2).

296 The operational semantics models executions under an arbitrary scheduler and client to
297 the overlay object $\texttt{F}$. An arbitrary client may have a thread call any operation available to it,
298 with any argument. Hence, each operational semantics rule non-deterministically chooses a
299 thread to take a step. This ensures we verify our modules as open components.

The states $(ths, s) \in \texttt{InterState}$ (interaction state) of the operational semantics are pairs of a *thread state* $ths \in \mathcal{T} \rightarrow \texttt{TState E F}$, which is a map from thread names to the

$$\textsc{OCall}\ \frac{ths(\alpha) = \texttt{Idle}}{(ths, s) \xrightarrow{\boldsymbol{\alpha}:\texttt{f(a)}} (ths[\alpha \mapsto \texttt{Cont f(a)}\ M^{\texttt{f(a)}}], s)} \qquad \textsc{ORet}\ \frac{ths(\alpha) = \texttt{Cont f(a) (Ret}\ v\texttt{)}}{(ths, s) \xrightarrow{\boldsymbol{\alpha}:\texttt{f(a)}.v} (ths[\alpha \mapsto \texttt{Idle}], s)}$$

$$\textsc{UCall}\ \frac{ths(\alpha) = \texttt{Cont f(a) (Vis}\ g(b)\ k\texttt{)} \qquad s \rightarrow^{\boldsymbol{\alpha}:\texttt{g(b)}}_{\texttt{V_E}} s'}{(ths, s) \xrightarrow{\boldsymbol{\alpha}:\texttt{g(b)}} (ths[\alpha \mapsto \texttt{UCall f(a) g(b)}\ k], s')} \qquad \textsc{URet}\ \frac{ths(\alpha) = \texttt{UCall f(a) g(b)}\ k \qquad s \rightarrow^{\boldsymbol{\alpha}:\texttt{g(b)}.v}_{\texttt{V_E}} s'}{(ths, s) \xrightarrow{\boldsymbol{\alpha}:\texttt{g(b)}.v} (ths[\alpha \mapsto \texttt{Cont f(a)}\ k(v)], s')}$$

$$\textsc{Silent}\ \frac{ths(\alpha) = \texttt{Cont f(a) (Tau}\ p\texttt{)}}{(ths, s) \xrightarrow{\epsilon} (ths[\alpha \mapsto \texttt{Cont f(a)}\ p], s)}$$

Figure 3 Operational Semantics

continuation of that thread, and an *underlay LTS state* $s \in S$. The continuation of each thread is defined as

$$\texttt{TState E F} := \texttt{Idle} \mid \texttt{Cont f(a)}\ p \mid \texttt{UCall f(a) g(}b\texttt{)}\ k$$

where $f \in \texttt{F}$, $a \in \texttt{par}(f)$, $p \in \texttt{Prog E ar}(f)$, $g \in \texttt{E}$, $b \in \texttt{par}(g)$, $k : \texttt{ar}(g) \to \texttt{Prog E ar}(f)$. We use $ths[\alpha \mapsto p]$ as the notation for updating the value of α to p in the map ths.

Fig. 3 defines the operational semantics rules. OCall selects an idle thread, non-deterministically chooses an operation $\texttt{f} \in \texttt{F}$ and an argument $\texttt{a} \in \texttt{par(f)}$, and changes the continuation for α to the code for $\texttt{f(a)}$ defined as $M^{\texttt{f(a)}}$. UCall locates a thread α whose next step in the continuation is an underlay call $\texttt{g(b)}$, performs the call step $\boldsymbol{\alpha}:\texttt{g(b)}$ in the underlay object $\texttt{V_E}$, and updates the underlay LTS state to s' accordingly. URet is analogous, except that it takes a return transition and continues with the continuation as specified by k. ORet matches on a thread whose continuation indicates it is ready to return from the overlay operation, and performs the overlay return by updating the continuation to $\texttt{Idle}$. Silent simply skips a $\texttt{Tau}$ with no change to the state.

We then define the concrete LTS of running M on top of $\texttt{V_E}$ as $\texttt{V_E} \rhd M : \texttt{Spec F} := (\texttt{InterState F V_E}, \rightarrow_{\texttt{V_E}\rhd M}, (ths_0, s_0))$, where ths_0 is the constant mapping to $\texttt{Idle}$ and $m \in \texttt{TEvent F}$, and transitions are given by the following relation

$$(ths, s) \rightarrow^{m}_{\texttt{V_E}\rhd M} (ths', s') \iff \exists ths'' \in \mathcal{T} \to \texttt{TState E F}, s'' \in S, t \in (\texttt{TEvent E})^*.$$
$$(ths, s) \xrightarrow{m} (ths'', s'') \wedge (ths'', s'') \xrightarrow{t} (ths', s')$$

Notice that only the overlay events $m \in \texttt{TEvent F}$ are visible in $\texttt{V_E} \rhd M$. The definition allows for a visible step m followed by several steps in the underlay object $\texttt{V_E}$, as enforced by the requirement that $t \in (\texttt{TEvent E})^*$. The underlay event trace is hidden behind an existential quantifier, making these implicit silent steps.

The following are some important properties of the operation - $\rhd$ -. The first two properties are monotonicity of vertical composition w.r.t. $\approx$ and $\sqsupseteq$, while the third property encodes both associativity of $\rhd$ and $:\rhd$, as well as compatibility of the two operations.

$$(1)\ M \approx M' \Rightarrow \texttt{V} \rhd M \equiv \texttt{V} \rhd M' \qquad (2)\ \texttt{V}' \sqsupseteq \texttt{V} \Rightarrow \texttt{V}' \rhd M \sqsupseteq \texttt{V} \rhd M$$

$$(3)\ \texttt{V} \rhd (M :\rhd N) \equiv (\texttt{V} \rhd M) \rhd N$$

3.7 Horizontal Composition

Both modules and specifications can be horizontally composed. For instance, given modules $M_L : \texttt{Mod E}_L\ \texttt{F}_L$ and $M_R : \texttt{Mod E}_R\ \texttt{F}_R$ we can define their horizontal composition $M_L + M_R$:

323 $\texttt{Mod}~(\texttt{E}_L + \texttt{E}_R)~(\texttt{F}_L + \texttt{F}_R)$, similarly to how we did with effect signatures, as the union of the
324 two collections:

325 $$(\texttt{M_L} + \texttt{M_R})^{f'} := \begin{cases} \texttt{mapProg}\,(\lambda x.\texttt{inl}~x)~M_L^f & f' = \texttt{inl}~f \\ \texttt{mapProg}\,(\lambda x.\texttt{inr}~x)~M_R^f & f' = \texttt{inr}~f \end{cases}$$

326 where $\texttt{mapProg}~(\lambda x.\texttt{inl}~x)$ replaces every event $\texttt{Vis}~e$ in M_L^f by $\texttt{Vis}~(\texttt{inl}~e)$ (and similarly
327 for the other branch).

For specifications $\texttt{V_L} : \texttt{Spec}~\texttt{E}_L$ and $\texttt{V_R} : \texttt{Spec}~\texttt{E}_R$ we can define a specification $\texttt{V_L} \otimes \texttt{V_R}$ given by the (locally sequential) asynchronous product of the two transition systems. We refer the reader to the mechanization for the definition. Below, we show the key compositional properties of $- \otimes -$:

$$\frac{\texttt{V_L} \sqsupseteq \texttt{V_L}' \qquad \texttt{V_R} \sqsupseteq \texttt{V_R}'}{\texttt{V_L} \otimes \texttt{V_R} \sqsupseteq \texttt{V_L}' \otimes \texttt{V_R}'} \qquad \frac{\texttt{V_L} \otimes \texttt{V_R} \sqsupseteq \texttt{V_L}' \otimes \texttt{V_R}'}{\texttt{V_L} \sqsupseteq \texttt{V_L}'} \qquad \frac{\texttt{V_L} \otimes \texttt{V_R} \sqsupseteq \texttt{V_L}' \otimes \texttt{V_R}'}{\texttt{V_R} \sqsupseteq \texttt{V_R}'}$$

$$\texttt{idM}_\texttt{E} + \texttt{idM}_\texttt{F} = \texttt{idM}_{\texttt{E}+\texttt{F}}$$

$$(\texttt{V_L} \rhd \texttt{M_L}) \otimes (\texttt{V_R} \rhd \texttt{M_R}) \equiv (\texttt{V_L} \otimes \texttt{V_R}) \rhd (\texttt{M_L} + \texttt{M_R})$$

328 The first three properties show that horizontal composition defines an order-isomorphism
329 for specification refinement. The last two properties show that the operation is functorial.

330 4 Linearizability and Possibilities

331 We now define the notion of linearizability used in the paper (§4.2), which is based on the
332 compositional linearizability formulation by [26]. Then, we go over how we implement one of
333 the main techniques for proving linearizability: possibility reasoning (§4.4).

334 4.1 Background

335 Safety for sequential objects is usually given by functional correctness. For instance, a counter
336 object provides operations $\texttt{get} : \mathbf{1} \to \mathbb{N}$, taking unit as argument and returning some natural
337 number, and $\texttt{inc} : \mathbf{1} \to \mathbf{1}$, taking unit as argument and returning unit. An implementation
338 of a counter is correct as long as `get` always returns the number of previous `inc` operations.
339 This means that given the trace:

340 $$s = \texttt{inc} \cdot \texttt{ok} \cdot \texttt{get}$$

341 of a correct counter implementation, we are guaranteed that the next event is a return of 1
342 to the operation `get`. However, for a concurrent counter implementation, given the trace:

343 $$\alpha_1{:}\texttt{inc} \cdot \alpha_1{:}\texttt{ok} \cdot \alpha_1{:}\texttt{get}$$

344 any number of increment operations by other threads might take effect:

345 $$\alpha_1{:}\texttt{inc} \cdot \alpha_1{:}\texttt{ok} \cdot \alpha_1{:}\texttt{get} \cdot \alpha_2{:}\texttt{inc} \cdot \alpha_2{:}\texttt{ok} \ldots \alpha_k{:}\texttt{inc} \cdot \alpha_k{:}\texttt{ok} \cdot \alpha_1{:}n$$

346 so that n might be any number between 1 and k. What makes a concurrent counter object
347 correct is therefore a more challenging question, and expressing “functional” correctness for
348 concurrent objects remained an open question until the 90s, when linearizability [15] was
349 proposed.

Linearizability asks that every trace generated by the concurrent counter implementation, such as trace s above, be *linearizable* with respect to some trace of a correct sequential counter. This is formalized by defining a partial order called *happens-before*, which associates to a trace s the partial order $\mathsf{hb}(s)$ defined as the smallest partial order such that $e \prec_{\mathsf{hb}(s)} e'$ whenever e and e' are events by the same thread and e appears before e' in s, or when e is a return event and e' is an invocation event. A trace s is then said to be linearizable w.r.t. an atomic trace t when there is a way to complete some pending invocations of s (by adding extra return events), then remove all remaining pending invocations, so as to obtain a trace s' satisfying that $\mathsf{hb}(s')$ is refined by $\mathsf{hb}(t)$ (in the sense that if $e \prec_{\mathsf{hb}(s')} e'$ then $e \prec_{\mathsf{hb}(t)} e'$). Atomic means that every invocation is immediately followed by its return, or, equivalently, that t is locally sequential (every thread alternates invocations and returns) and $\prec_{\mathsf{hb}(t)}$ is a linear order. An implementation is then said to be linearizable w.r.t. a specification if all traces generated by the implementation are linearizable w.r.t. the specification.

The requirement that t, and consequently specifications, be atomic ensures that concurrent data structures such as stacks, queues, and counters are all specified by essentially the same specification as the corresponding sequential data structures. This requirement, however, proves too restrictive, as many reasonable concurrent objects admit no corresponding sequential object. For example, the exchanger in §2 is a standard synchronization primitive (for instance, available as part of `Java`'s standard libraries [4]). Nonetheless, it admits no reasonable atomic specification, intuitively because an exchange can only happen if two threads are running concurrently. To remedy this, *set linearizability* was proposed [24]. Set linearizability only modifies the definition of linearizability by requiring that the traces of the specification be of the form:

$$I_1 \cdot R_1 \cdot I_2 \cdot R_2 \cdot \ldots \cdot I_n \cdot R_n$$

where each I_k is a sequence of invocations and each R_k is a sequence of responses containing *all* and *only* the responses to the invocations in I_k (except for R_n, which may contain only *some* such responses). In addition, one requires that if

$$I_1 \cdot R_1 \cdot I_2 \cdot R_2 \cdot \ldots \cdot I_n \cdot R_n$$

is in the specification, then so is

$$I'_1 \cdot R'_1 \cdot I'_2 \cdot R'_2 \cdot \ldots \cdot I'_n \cdot R'_n$$

where each I'_k (resp. R'_k) is some reordering of I_k (resp. R_k). This means that we may see these traces as sequences of sets of invocations, each followed by the set of returns to that set of invocations.

This modification to linearizability allows it to specify many concurrent objects whose behaviors involve some kind of synchronization, such as exchangers, barriers, and blocking stacks (stacks where `pop` blocks on the empty stack). We will see that it also helps in specifying objects beyond synchronization primitives, such as the `TryStack`. However, there are concurrent objects and certain distributed tasks whose consistency is even weaker than that provided by set linearizability. For example, it is possible to implement a write-snapshot object [3][2] validating the trace:

$$s' = \boldsymbol{\alpha}{:}\mathtt{ws}(3) \cdot \boldsymbol{\beta}{:}\mathtt{ws}(5) \cdot \boldsymbol{\alpha}{:}\mathtt{ws}(3).\{3,5\} \cdot \boldsymbol{\gamma}{:}\mathtt{ws}(2) \cdot \boldsymbol{\beta}{:}\mathtt{ws}(5).\{3,5,2\}$$

[2] A write-snapshot provides a single operation `ws` that takes a value as argument and returns a set of written values as output.

Here, when α takes its snapshot, β's value of 5 is already visible, but by the time β manages to take its own snapshot, γ's value is now visible too. Because of this, the return to β's call cannot happen together with α's return, and hence set linearizability cannot capture this object's consistency.

To accommodate such objects, interval linearizability [5] was proposed, which further relaxes the notion of specification in linearizability's definition so that in a trace

$$I_1 \cdot R_1 \cdot I_2 \cdot R_2 \cdot \ldots \cdot I_n \cdot R_n$$

R_k is now allowed to contain returns to pending invocations from I_l for any $l \leq k$, and in particular it may contain only *some* of the returns in any I_l (including I_k). The requirement that reorderings within I_k or R_k are all valid remains. This has been shown to be a *complete* correctness criterion for distributed tasks [5, 12]. Interval linearized specifications therefore allow traces with a significant degree of concurrency, such as s' above. Nonetheless, these specifications may still be simpler than the traces generated by the implementation. The write-snapshot we discuss and verify in our mechanization admits an interval linearized specification that is deterministic in the sense that given a trace:

$$I_1 \cdot R_1 \cdot \ldots \cdot I_k$$

then in any two extensions

$$I_1 \cdot R_1 \cdot \ldots \cdot I_k \cdot R_k \qquad \text{and} \qquad I_1 \cdot R_1 \cdot \ldots \cdot I_k \cdot R_k'$$

where R_k and R_k' contain only responses, if $\boldsymbol{\alpha}{:}\mathtt{ws}(v).S$ appears in R_k then either $\boldsymbol{\alpha}{:}\mathtt{ws}(v).S$ also appears in R_k' or R_k' can be extended with $\boldsymbol{\alpha}{:}\mathtt{ws}(v).S$. In other words, any returns that can happen for any given thread are unique up to the ordering of the responses. This makes the linearized specification significantly easier to work with for a client of the write-snapshot object than using all the traces generated by the implementation, which present more concurrency and non-determinism.

Nonetheless, interval linearizability still does not accommodate all concurrent objects. This last issue is more subtle, and we recommend that the interested reader consult [26] for a more thorough treatment. Succinctly, the requirement that *all* pending invocations that have not been completed must be removed in the linearization is too strong for certain concurrent objects. There are circumstances where an invocation is *detectable* to other threads even though its response is still not enabled. Compositional linearizability [25, 26] further weakens interval linearizability by allowing some invocations to be completed, some to be removed, and some to remain in the linearization. This makes compositional linearizability complete for concurrent objects. However, perhaps the main advance of compositional linearizability is that it gives a novel treatment of the theory of linearizability showing that, within a compositional model of computation, linearizability can be characterized entirely in terms of composition and refinement. In this paper, we adopt compositional linearizability as our notion of correctness and mechanize many aspects of the comprehensive theory of [26]. In particular, they present rather algebraic proofs of all the key properties of linearizability based on a different (but equivalent) formulation that circumvents the use of partial orders or rewrite systems. We have found that this reformulation makes mechanization tractable despite the significantly more challenging setting of working with non-atomic and potentially blocking specifications. At the same time, we find it productive to formulate their ideas using LTSs rather than strategies (final coalgebras for these LTSs). To do so, we made several improvements to their proof technique based on possibilities. The remainder of this section concerns our mechanization of compositional linearizability and possibilities, ultimately leading to our proof technique in §5.

4.2 Linearizability

The observation at the core of [26] is that, in the context of a compositional model for computation, the linearizability theory can be entirely derived from the definition of the identity for composition of modules. They refer to their notion of linearizability as *compositional linearizability* and demonstrate that it is a conservative generalization of Herlihy-Wing linearizability [15], set linearizability [24], and interval linearizability [5], as it becomes equivalent to these criteria under appropriate restrictions.

In our mechanization, we choose to formalize compositional linearizability using its refinement-based definition, which can be defined using `idM` as follows:

▶ **Definition 3.** *Given a specification* $V : \texttt{Spec}\ E$*, we define* $K\ V : \texttt{Spec}\ E$ *by* $K\ V := V \rhd \texttt{idM}$*.*

We say a concrete LTS $V' : \texttt{Spec}\ E$ *is linearizable w.r.t.* $V : \texttt{Spec}\ E$*, written* $V' \rightsquigarrow V$ *when* $V' \sqsupseteq K\ V$*.*

Intuitively, the reason why this definition agrees with the usual linearizability definition is that the set of behaviors of `K V` is exactly the set of all traces linearizable w.r.t. some behavior of `V`. The operations of `E` made by the overlay to `idM` form a bracket around the corresponding underlay operations. Because of this, happens-before ordering is preserved from the overlay of `idM` to its underlay. The fact that pending calls may be removed or completed in a trace corresponds to the fact that in the operational semantics of $\texttt{V} \rhd \texttt{idM}$, one can be in a state where the call has been made in the overlay of `idM` but not in the underlay yet. There are states where a return has been made to the underlay operation in `idM` but has not returned to the overlay yet. Therefore, `K V` defined above contains all traces linearizable w.r.t. the specification. We refer readers to [26] for a detailed treatment of how this definition relates to previous definitions of linearizability.

Our definition of linearizability enjoys observational refinement and locality (Theorem 4). They follow from generalizing the proofs of compositional linearizability from [26] to handle our more heterogeneous compositional structure, discussed in §3.

▶ **Theorem 4** (Observational Refinement and Locality). *For any* $\texttt{V_E}', \texttt{V_E} \in \texttt{Spec}\ E$*,* $\texttt{V_F}', \texttt{V_F} \in \texttt{Spec}\ F$*, and* $M \in \texttt{Mod}\ E\ G$*, we have observational refinement (1) and locality (2):*

(1) Behaviors of running any module M *above the concrete LTS* $\texttt{V_E}'$ *are contained in those of running the same* M *above the linearized specification* $\texttt{V_E}$*, i.e.,* $\texttt{V_E}' \rightsquigarrow \texttt{V_E} \Longrightarrow (\texttt{V_E}' \rhd M) \sqsupseteq (\texttt{V_E} \rhd M)$*.*

(2) A system of multiple objects is linearizable iff all of these objects are separately linearizable, i.e., $\texttt{V_E}' \rightsquigarrow \texttt{V_E} \wedge \texttt{V_F}' \rightsquigarrow \texttt{V_F} \iff \texttt{V_E}' \otimes \texttt{V_F}' \rightsquigarrow \texttt{V_E} \otimes \texttt{V_F}$*.*

4.3 Compositionality

Through compositional linearizability [26], LHL gains horizontal and vertical composition without any additions to the logic. LHL's soundness provides linearizability proofs of different modules, which can be horizontally and vertically composed together to obtain the linearizability of a larger system. These results are obtained by combining locality and observational refinement with the compositional structure of the model defined in §3.

▶ **Theorem 5** (Horizontal Comp.). *If* $(\texttt{V_E}_1 \rhd \texttt{M_F}_1) \rightsquigarrow \texttt{V_F}_1$ *and* $(\texttt{V_E}_2 \rhd \texttt{M_F}_2) \rightsquigarrow \texttt{V_F}_2$*, then* $((\texttt{V_E}_1 \otimes \texttt{V_E}_2) \rhd (\texttt{M_F}_1 \otimes \texttt{M_F}_2)) \rightsquigarrow (\texttt{V_F}_1 \otimes \texttt{V_F}_2)$*.*

▶ **Theorem 6** (Vertical Comp.). *If* $(\texttt{V_E} \rhd \texttt{M_E}) \rightsquigarrow \texttt{V_F}$ *and* $(\texttt{V_F} \rhd \texttt{M_F}) \rightsquigarrow \texttt{V_G}$*, then* $(\texttt{V_E} \rhd (\texttt{M_E} :\rhd \texttt{M_F})) \rightsquigarrow \texttt{V_G}$*.*

480 Horizontal composition enables importing multiple objects together through the tensor
481 operator ($\otimes$) and using them concurrently. The locality of compositional linearizability
482 guarantees that both objects will preserve their original specifications, and users of both
483 objects can enjoy the interface $\mathtt{F}_1 + \mathtt{F}_2$ with both sets of operations available.

484 Vertical composition ($\triangleright$) "links" the overlay object's code with the imported underlay
485 object's code. Since vertical composition ($\triangleright$) is compatible with linking ($:\triangleright$), the observational
486 refinement [26] ensures running the linked code `M_E` $:\triangleright$ `M_F` above the lower-level specification
487 `V_E'` still produces traces linearizable to the top-level specification `V_F`.

488 This is how we "glue" together the individual verified components in Fig. 1 into the
489 whole implementation of the EBStack to obtain the correctness of the entire system.

490 4.4 Possibility Reasoning

491 To prove linearizability, we employ *possibility reasoning*. [26] have shown a bisimulation
492 between possibilities and their formulation of the operational semantics of `idM` running on
493 top of `V_F`, implying that possibility reasoning is sound and complete for compositional
494 linearizability proofs. In our setting, we redesign the possibility reasoning technique to
495 contend with our use of state as explained in §1.

496 Assuming we want to show M running above `V_E` is linearizable w.r.t. `V_F`, i.e., (`V_E` $\triangleright$
497 M) $\rightsquigarrow$ `V_F`, a possibility $\rho \in$ `Poss` is defined as a triple $\langle \rho_S, \rho_C, \rho_R \rangle$ of (1) the linearized
498 state ρ_S in `V_F`, (2) a partial map ρ_C from thread names $\alpha \in \mathcal{T}$ to call events of `F`, and (3) a
499 partial map ρ_R from thread names $\alpha \in \mathcal{T}$ to return events of `F`. We write $\rho_C(\alpha)$ for the event
500 that ρ_C maps α to; $\rho_C(\alpha) = \bot$ when it is undefined; and $\rho_C[\alpha \mapsto m]$ for substitution of m
501 for $\rho_C(\alpha)$; and similarly for ρ_R. The mapping ρ_C stores pending calls that have happened
502 in the concrete execution but have not been performed in the linearized state s; ρ_R stores
503 returns that have been made in the linearized state ρ_S, but not in the concrete execution.
504 Therefore, from the perspective of a particular thread α, a possibility can be in one of four
505 states:

506 **Idle:** The last transition by α in `V_F` was a return, and both $\rho_C(\alpha) = \bot$ and $\rho_R(\alpha) = \bot$.

507 **Pending Call:** α's last transition in `V_F` was a return, $\rho_C(\alpha) = \mathtt{m}$ for some $m \in \mathtt{F}$ and
508 $\rho_R(\alpha) = \bot$.

509 **Call Done:** The last transition by α in `V_F` was a call $\boldsymbol{\alpha}{:}m$, $\rho_C(\alpha) = \mathtt{m}$ and $\rho_R(\alpha) = \bot$.

510 **Pending Return:** The last transition by α in `V_F` was a return $\boldsymbol{\alpha}{:}m.n$, $\rho_C(\alpha) = \mathtt{m}$ and
511 $\rho_R(\alpha) = \mathtt{m.n}$.

512 We formalize possibility updates, the axioms to manipulate possibilities, as a transition
513 system whose states are possibilities over `V_F` and transitions are given by:

$$\textsc{Invoke}\ \frac{\rho_C(\alpha) = \bot \qquad \rho_R(\alpha) = \bot}{\langle \rho_S, \rho_C, \rho_R \rangle \dashrightarrow \langle \rho_S, \rho_C[\alpha \mapsto m], \rho_R \rangle} \qquad \textsc{Return}\ \frac{\rho_C(\alpha) = m \qquad \rho_R(\alpha) = n}{\langle \rho_S, \rho_C, \rho_R \rangle \dashrightarrow \langle \rho_S, \rho_C[\alpha \mapsto \bot], \rho_R[\alpha \mapsto \bot] \rangle}$$

$$\textsc{CCommit}\ \frac{\rho_S \xrightarrow{\boldsymbol{\alpha}:m}_{\mathtt{V_F}} t \qquad \rho_C(\alpha) = m \qquad \rho_R(\alpha) = \bot}{\langle \rho_S, \rho_C, \rho_R \rangle \dashrightarrow \langle t, \rho_C, \rho_R \rangle} \qquad \textsc{RCommit}\ \frac{\rho_S \xrightarrow{\boldsymbol{\alpha}:m.n}_{\mathtt{V_F}} t \qquad \rho_C(\alpha) = m \qquad \rho_R(\alpha) = \bot}{\langle \rho_S, \rho_C, \rho_R \rangle \dashrightarrow \langle t, \rho_C, \rho_R[\alpha \mapsto n] \rangle}$$

$$\overline{\rho} \twoheadrightarrow \overline{\sigma} \iff \forall \sigma \in \overline{\sigma}.\, \exists \rho \in \overline{\rho}.\, \rho \dashrightarrow^* \sigma$$

Figure 4 Possibility update rules

$$
\begin{array}{ll}
 & \langle ([], \bot), \varnothing, \varnothing \rangle \\
\dashrightarrow (\textsc{Invoke}) & \langle ([], \bot), [\alpha_1 \mapsto \texttt{push(1)}], \varnothing \rangle \\
\dashrightarrow (\textsc{Invoke}) & \langle ([], \bot), [\alpha_1 \mapsto \texttt{push(1)}][\alpha_2 \mapsto \texttt{push(2)}], \varnothing \rangle \\
\dashrightarrow (\textsc{CCommit}) & \langle ([], \boldsymbol{\alpha_i}{:}\texttt{push(i)}]), [\alpha_1 \mapsto \texttt{push(1)}][\alpha_2 \mapsto \texttt{push}(2)], \varnothing \rangle \\
\dashrightarrow (\textsc{RCommit}) & \left\langle (i :: [], \bot), \begin{bmatrix} \alpha_1 \mapsto \texttt{push(1)}, \\ \alpha_2 \mapsto \texttt{push(2)} \end{bmatrix}, [\alpha_i \mapsto \texttt{push(i).()}] \right\rangle \\
\dashrightarrow (\textsc{CCommit}) & \left\langle \begin{pmatrix} i :: [], \\ \boldsymbol{\alpha_j}{:}\texttt{push(j)}] \end{pmatrix}, \begin{bmatrix} \alpha_1 \mapsto \texttt{push(1)}, \\ \alpha_2 \mapsto \texttt{push(2)} \end{bmatrix}, [\alpha_i \mapsto \texttt{push(i)}.()] \right\rangle \\
\dashrightarrow (\textsc{RCommit}) & \left\langle (j :: i :: [], \bot), \begin{bmatrix} \alpha_1 \mapsto \texttt{push(1)}, \\ \alpha_2 \mapsto \texttt{push(2)} \end{bmatrix}, \begin{bmatrix} \alpha_i \mapsto \texttt{push(i).()}, \\ \alpha_j \mapsto \texttt{push}(j).() \end{bmatrix} \right\rangle \\
\dashrightarrow (\textsc{Return}) & \langle (j :: i :: [], \bot), [\alpha_j \mapsto \texttt{push(j)}], [\alpha_j \mapsto \texttt{push(j).()}] \rangle \\
\dashrightarrow (\textsc{Return}) & \langle (j :: i :: [], \bot), \varnothing, \varnothing \rangle \\
\dashrightarrow (\textsc{Invoke}) & \langle (j :: i :: [], \bot), [\alpha_3 \mapsto \texttt{pop()}], \varnothing \rangle \\
\dashrightarrow (\textsc{CCommit}) & \langle (j :: i :: [], \boldsymbol{\alpha_3}{:}\texttt{pop()}), [\alpha_3 \mapsto \texttt{pop()}], \varnothing \rangle \\
\dashrightarrow (\textsc{RCommit}) & \langle (i :: [], \bot), [\alpha_3 \mapsto \texttt{pop()}], [\alpha_3 \mapsto \texttt{pop().j}] \rangle
\end{array}
$$

Figure 5 Example of possibility-based linearizability proof

Intuitively, a possibility encodes a proof of linearizability. The Invoke and Return model the events of the concrete trace, and CCommit and RCommit the corresponding events in the linearization. The axioms on possibility updates ensure that as long as the initial possibility is valid, the updated possibility always corresponds to some *valid* linearization, i.e., (1) only call events that happened in the concrete execution can be added to the possibility, (2) only transitions respecting local sequentiality can be made to the possibility, and (3) return events missing in the concrete execution can be added to complete pending calls in the possibility. The remaining obligation to show the consistency of linearized return values and actual returned values will be addressed by the program logic. For instance, a possible proof that the trace t' below is linearizable w.r.t. $t_{i,j}$ corresponds to the sequence of possibility updates shown in Fig. 5.

$$
\begin{aligned}
t' &= \boldsymbol{\alpha_1}{:}\text{push}(1) \cdot \boldsymbol{\alpha_2}{:}\text{push}(2) \cdot \boldsymbol{\alpha_1}{:}\text{push}(1).() \cdot \boldsymbol{\alpha_2}{:}\text{push}(2).() \cdot \boldsymbol{\alpha_3}{:}\texttt{pop()} \\
t_{i,j} &= \boldsymbol{\alpha_i}{:}\text{push}(i) \cdot \boldsymbol{\alpha_i}{:}\text{push}(i).() \cdot \boldsymbol{\alpha_j}{:}\text{push}(j) \cdot \boldsymbol{\alpha_j}{:}\text{push}(j).() \cdot \boldsymbol{\alpha_3}{:}\texttt{pop()} \cdot \boldsymbol{\alpha_3}{:}\texttt{pop()}.j
\end{aligned}
$$

When verifying a stack implementation, we can determine if the correct linearization is $t_{1,2}$ or $t_{2,1}$ by inspecting the underlay state. If we focus on Invoke and Return events, we can reconstruct the concrete trace t', and if we focus on CCommit and RCommit, we can rebuild the linearized trace $t_{i,j}$.

Single possibilities are not sufficient in general when constructing proofs of future-dependent linearizations inductively. We must maintain multiple possibilities simultaneously to contend with current possibilities being invalidated in the future. To accomplish this, we lift the possibility axioms from a single possibility ρ to a non-empty *set* of possibilities $\overline{\rho}$, and define the lifted transition $- \twoheadrightarrow -$.[3] In other words, a set of possibilities $\overline{\rho}$ can be updated

[3] Formally, we have described how to construct the equivalent of forward simulations. Using a power-set construction, we obtain the equivalent of a forward-backward simulation, which is complete for trace equivalence [23].

536 into $\overline{\sigma}$ as long as every possibility in $\overline{\sigma}$ is reachable after zero or finitely many valid updates
537 from some possibility in $\overline{\rho}$.

538 In the stack example above, if the stack's linearizability proof were future-dependent
539 (e.g., the timestamped stack [8]), we would maintain a set $\overline{\rho} = \{\rho_{1,2}, \rho_{2,1}\}$ for the trace t'.
540 The possibility $\rho_{1,2}$ corresponds to the case $i = 1$ and $j = 2$, and $\rho_{2,1}$ the other case. We
541 would then update $\overline{\rho}$ according to the updates in the example above for the corresponding
542 assignments of i and j. When the `pop()` operation reaches its linearization point, we would
543 find out which of the two pushes was deemed to have occurred first, and remove whichever
544 of $\rho_{1,2}$ or $\rho_{2,1}$ has been invalidated.

545 5 Program Logic

546 Now we are ready to present our program logic (§5.1-§5.6) and its soundness and completeness
547 (§5.7). Our program logic, building on that of [26], is a Rely-Guarantee program logic. The
548 main idea of rely-guarantee reasoning [17] is that each thread over-approximates its local
549 steps in terms of a *guarantee* relation $\mathcal{G}$, and over-approximates its environment steps (steps
550 by other threads) in terms of a *rely* relation $\mathcal{R}$. One then shows that the thread guarantees
551 its steps follow $\mathcal{G}$, while relying on its environment following $\mathcal{R}$. The main aspect of rely-
552 guarantee reasoning is the notion of *stability*, which is what enables parallel composition to
553 become a congruence.

Stability is the notion that when a thread shows that steps satisfy a predicate P, such as pre-conditions, invariants or post-conditions, these predicates must be shown to be stable with respect to $\mathcal{R}$, in the sense that:

$$\mathcal{R}; P \implies P \qquad\qquad P; \mathcal{R} \implies P$$

554 that is, predicates are invariant upon the environment taking steps *before* or *after* the
555 predicates (where $-;-$ stands for relational composition). When a predicate satisfies this, we
556 say it is *stable* w.r.t. $\mathcal{R}$, written $\mathtt{Stable}(\mathcal{R}, P)$. Stability ensures that any predicates we verify
557 a thread's local steps to satisfy will not be invalidated by having the scheduler introduce
558 steps by other threads between local steps, and it is fundamental to ensuring threads may be
559 appropriately parallel composed later.

Then, if a thread α is verified against $\mathcal{R}[\alpha]$ and $\mathcal{G}[\alpha]$, and β is verified against $\mathcal{R}[\beta]$ and $\mathcal{G}[\beta]$, we are allowed to parallel compose them when

$$\mathcal{G}[\alpha] \implies \mathcal{R}[\beta] \qquad \text{and} \qquad \mathcal{G}[\beta] \implies \mathcal{R}[\alpha]$$

560 which ensures that α and β follow each other's assumptions about their environments.

561 5.1 Proof State and Assertions

562 In LHL, a proof configuration is a tuple $(s, \overline{\rho})$ of a concrete LTS state $s \in \mathtt{InterState}$ and
563 a set of possibilities $\overline{\rho} \in \mathcal{P}(\mathtt{Poss})$. The component s represents the current state of the
564 program, which follows the actual semantics of the program, while $\overline{\rho}$ represents linearized
565 overlay LTS states, which are managed by the prover following the possibility update rules
566 in §4.2.

567 By using a set of possibilities instead of a single possibility, users of LHL can track different
568 linearizations corresponding to the current state. They can remove extra possibilities later
569 when future events invalidate these linearizations. This allows us to support objects (such as

570 the Herlihy-Wing queue [15]) with future-dependent linearizations [31], which is essential for
571 the completeness proof of LHL.
572 We define preconditions P as predicates over the proof state, while rely/guarantee
573 conditions $\mathcal{R}, \mathcal{G}$ and postconditions Q are defined as relations over the state. We use binary
574 postconditions relating pre-/post-states to make certain proofs easier.

575 5.2 The Linearizability Hoare Logic

576 LHL consists of three judgments: (1) the module judgment $V_E, V_F, \mathcal{R}, \mathcal{G}, P, Q \vDash M$ groups
577 all side conditions and implies our linearizability condition ($\texttt{V_E} \triangleright \texttt{M_F}) \rightsquigarrow \texttt{V_F}$, (2) the
578 program judgment $\mathcal{R}, \mathcal{G} \vdash \{P\}\, e\, \{Q\}$ verifies the correctness of the method body, and (3) the
579 commit judgment $\mathcal{G} \vdash_\alpha \{P\}\, m\, \{Q\}$ handles individual underlay events and applies possibility
580 updates.

581 5.3 Module Judgment

To establish module correctness, the prover needs to find pre-/post-conditions for each $f \in \texttt{F}$
and rely/guarantee relations, and show the following:

$$\frac{\begin{array}{c}\forall \alpha, s, \overline{\rho}.\, (s, \overline{\rho})\ \mathcal{R}[\alpha]\ (s, \overline{\rho}) \\ \forall \alpha.\, \mathcal{R}[\alpha]; \mathcal{R}[\alpha] \implies \mathcal{R}[\alpha] \qquad \forall \alpha, \beta.\, \alpha \neq \beta \wedge \mathcal{G}[\alpha] \cup \texttt{Inv}[\alpha](-) \cup \texttt{Ret}[\alpha](-) \implies \mathcal{R}[\beta] \\ \forall \alpha, f.\, \texttt{Stable}(\mathcal{R}[\alpha], P[\alpha]_f) \qquad \forall \alpha, f.\, P[\alpha]_f([], \langle \texttt{init(V_F)}, [], [] \rangle) \\ \forall \alpha, f.\, \mathcal{R}[\alpha], \mathcal{G}[\alpha] \vdash \{P[\alpha]_f; \texttt{Inv}[\alpha](f)\}\, M[\alpha]^f\, \{Q[\alpha]_f\} \qquad \forall \alpha, f, g.\, P[\alpha]_f; Q[\alpha]_f \implies P[\alpha]_g\end{array}}{\texttt{V_E}, \texttt{V_F}, \mathcal{R}, \mathcal{G}, P, Q \vDash M}$$

582 where $\texttt{Inv}(-) := \cup_{f \in F} \texttt{Inv}(f)$, $\texttt{Ret}(-) := \cup_{f \in F} \texttt{Ret}(f)$. The first few conditions are standard
583 rely-guarantee conditions, respectively: that $\mathcal{R}[\alpha]$ is reflexive and self-stable, that threads
584 respect each other's relies, and that method pre-conditions are stable. Other than the
585 standard rely-guarantee side conditions, we require for all methods that (1) the initial proof
586 state satisfies its pre-condition, (2) the required Hoare triples can be established for it, and
587 (3) its post-condition implies the pre-condition of any method, so the system can still run
588 safely afterwards.

589 ▶ **Example 7** (Verifying the Exchanger Object)**.** For the exchanger, we only need to find one
590 pre-/post-condition for the `exch` method, which we made into an invariant I that holds at
591 any step in all threads. It consists of three disjuncts that hold at different stages of the
592 exchanger: when a thread makes an exchange offer, when the other thread accepts the offer,
593 and when the offering thread clears the offer. They mainly relate the concrete state to
594 the linearized state. For simplicity, we use a singleton possibility, as no future-dependent
595 linearization is required here. For example, the **accepted** predicate requires the prover to
596 linearize the response $\rho_R(\alpha)$ to the `exch` method made by the offering thread α when the
597 CAS cell stores the `Acc` state.

598 $$I(s, \rho) := \exists \alpha, \beta, v, w.\, \textbf{offered}(\alpha, v)(s, \rho) \vee \textbf{accepted}(\alpha, \beta, v, w)(s, \rho) \vee \textbf{cleared}(s, \rho)$$

599 $$\textbf{offered}(\alpha, v)(s, \rho) := s_{\texttt{CAS}} = \texttt{Off}(\alpha, v) \wedge \rho_S = (\{\}, \{\}) \wedge \rho_C(\alpha) = \texttt{exch}(v) \wedge \rho_R(\alpha) = \bot$$

600 $$\textbf{accepted}(\alpha, \beta, v, w)(s, \rho) := s_{\texttt{CAS}} = \texttt{Acc}(\alpha, \beta, v, w) \wedge \rho_S = (\{\}, \{\})$$

601 $$\wedge\, \rho_C(\alpha) = \texttt{exch}(v) \wedge \rho_R(\alpha) = \texttt{exch}(v).w$$

602 $$\textbf{cleared}(s, \rho) := s_{\texttt{CAS}} = \bot \wedge \rho_S = (\{\}, \{\})$$

603 The invariant obviously satisfies all conditions except the rely-guarantee-related ones and
604 the Hoare triple, which we supply and prove later in this section.

605
5.4 Program Judgment

The program judgment consists of the following Hoare-style rules:

$$\frac{P \implies Q\,v}{\mathcal{R},\mathcal{G} \vdash \{P\}\,\mathtt{Ret}\,v\,\{Q\}}\ \textsc{Ret}$$

$$\frac{\mathtt{Stable}(\mathcal{R},Q_S) \qquad \mathcal{R},\mathcal{G} \vdash \{P;Q_S\}\,p\,\{Q\} \qquad P \xrightarrow{\alpha:\epsilon} \lambda s,t,\overline{\rho}.\,Q_S(s,\overline{\rho},t,\overline{\rho}) \wedge \mathcal{G}(s,\overline{\rho},t,\overline{\rho}))}{\mathcal{R},\mathcal{G} \vdash \{P\}\,\mathtt{Tau}\,p\,\{Q\}}\ \textsc{Silent}$$

$$\frac{\begin{array}{c}\mathcal{G} \vdash_\alpha \{P\}\,g(x)\,\{Q_C\} \qquad \forall v.\mathcal{G} \vdash_\alpha \{P;Q_C\}\,g(x).v\,\{Q_R(v)\} \\ \mathtt{Stable}(\mathcal{R},Q_C) \qquad \forall v.\,\mathtt{Stable}(\mathcal{R},Q_R(v)) \\ \forall v.\,\mathcal{R},\mathcal{G} \vdash \{P;Q_C;Q_R(v)\}\,k(v)\,\{Q\}\end{array}}{\mathcal{R},\mathcal{G} \vdash \{P\}\,\mathtt{Vis}\,g(x)\,k\,\{Q\}}\ \textsc{UnderlayStep}$$

$$\text{where } P \xrightarrow{e} Q \iff \forall s,t,\overline{\rho}.\,P(s,\overline{\rho}) \wedge s \xrightarrow{e} t \Rightarrow Q(s,t,\overline{\rho})$$

These core rules are applicable to all programs. In practice, though, the user will define control structures along with their corresponding *derived rules*, and the core rules will not be used directly. For instance, `bind` and `call` may be defined, and their derived rules are as follows.

$$\frac{\mathcal{R},\mathcal{G} \vdash \{P\}\,e\,\{Q\} \qquad \forall v.\mathcal{R},\mathcal{G} \vdash \{Q(v)\}\,k(v)\,\{R\}}{\mathcal{R},\mathcal{G} \vdash \{P\}\,v \leftarrow e;\,k(v)\,\{R\}}$$

$$\frac{\begin{array}{c}\mathtt{Stable}(\mathcal{R},Q) \qquad \mathtt{Stable}(\mathcal{R},R) \qquad \mathcal{G} \vdash_\alpha \{P\}\,g(x)\,\{Q\} \\ \forall v.\,\mathcal{G} \vdash_\alpha \{P;Q\}\,g(x).v\,\{R(v)\}\end{array}}{\mathcal{R},\mathcal{G} \vdash \{P\}\,g(x)\,\{P;Q;R\}}$$

606
5.5 Commit Judgment

607 The final component of the proof is the *commit judgment* $G \vdash_i \{P\}\,m\,\{Q\}$ generalizing
608 linearization points into possibility updates.

$$\frac{P \xrightarrow{\alpha:m} \lambda s,t,\overline{\rho}.\,\exists\overline{\sigma}.\overline{\rho} \twoheadrightarrow^* \overline{\sigma} \wedge Q(s,\overline{\rho},t,\overline{\sigma}) \wedge \mathcal{G}(s,\overline{\rho},t,\overline{\sigma})}{\mathcal{G} \vdash_\alpha \{P\}\,m\,\{Q\}}\ \textsc{Commit}$$
609

610 The commit judgment asks the following: for any state $(s,\overline{\rho})$ satisfying the pre-condition
611 P and any t reachable from s through the operational semantics, the user needs to produce
612 new linearizations $\overline{\sigma}$ reachable after finitely many possibility updates ($\overline{\rho} \twoheadrightarrow \overline{\sigma}$, defined in §4.4)
613 and ensure the postcondition and guarantee relation are satisfied. In LHL, these updates
614 are restricted to CCommit or RCommit; the necessary Invoke and Return updates are
615 applied automatically by the logic. The central proof obligation of LHL is to manage the set
616 of possibilities $\overline{\rho}$ and maintain a sufficient set of correct linearizations. The user uses the
617 commit judgment to update the linearization as they verify an operation, until they reach
618 the postcondition Q_f. After this, the user will have to show that the series of commits they
619 performed resulted in a linearization with the same return value as that produced by the
620 operational semantics.

621 ▶ **Example 8** (Verifying the Exchanger Object Cont.). We take line 5 in the exchanger code
622 in Fig. 1 as an example:

623 $\{\mathbf{offered}(\alpha,v)(s,\rho) \vee \exists\beta\,w.\mathbf{accepted}(\alpha,\beta,v,w)(s,\rho)\}$

624 $\mathtt{if}(C.\mathtt{cas}(\mathtt{Off}(\alpha,v),\bot))\,\{$

625 $\left\{\begin{array}{l}(\mathbf{cleared}(t,\sigma) \vee (\exists\beta\,w.\mathbf{offered}(\beta,w)(t,\sigma) \\ \quad \vee\exists\gamma\,u.\mathbf{accepted}(\beta,\gamma,w,u)(t,\sigma))) \wedge \rho_R(\alpha) = \mathtt{exch(v)}.\bot\end{array}\right\}$

Here, we only consider the success branch of the CAS operation, and for simplicity, we only use a singleton possibility. Per the commit judgment, we need to prove:

$$\forall s, \rho, t. \begin{pmatrix} (\textbf{offered}(\alpha, v)(s, \rho) \vee \exists \beta\, w. \textbf{accepted}(\alpha, \beta, v, w)(s, \rho)) \\ \wedge\ s \xrightarrow{\boldsymbol{\alpha}:\texttt{cas}(\texttt{Off}(\alpha,v)).\texttt{true},\bot} t \end{pmatrix}$$

$$\Rightarrow \exists \sigma. \begin{pmatrix} \rho \dashrightarrow^{\boldsymbol{\alpha}:\texttt{exch}(v),\boldsymbol{\alpha}:\texttt{exch}(v).\bot} \sigma \wedge \mathcal{G}(s, \rho, t, \sigma) \wedge \\ (\textbf{cleared}(t, \sigma) \vee (\exists \beta\, w. \textbf{offered}(\beta, w)(t, \sigma) \\ \vee \exists \gamma\, u. \textbf{accepted}(\beta, \gamma, w, u)(t, \sigma))) \end{pmatrix}$$

We observe that the transition from s to t makes the **accepted** case of the precondition impossible, so we are left with $\textbf{offered}(\alpha, v)$ as the precondition. We choose to use $(\{\}, \{\})$ as σ (the same as ρ) and perform the possibility update ($\rho \dashrightarrow \sigma$) by making two transitions with event labels $\boldsymbol{\alpha}:\texttt{exch}(v)$ and $\boldsymbol{\alpha}:\texttt{exch}(v).\bot$. The return value of this linearization is $\bot$, and the actual return value of this branch is also $\bot$, so this linearization is valid. This possibility update is also safe because no concurrent exchange operation will successfully exchange its value for v and the new proof state satisfies the **cleared** disjunct of the post-condition.

Formally, the following rely-guarantee definition justifies this update, which matches the $\mathcal{G}_{\textbf{fail}}$ relation.

$$\mathcal{G}_{\textbf{fail}}(\alpha, v)\ (s, \rho, t, \sigma) := \begin{pmatrix} s \xrightarrow{\boldsymbol{\alpha}:\texttt{cas}(\alpha,\bot).\texttt{true}} t \\ \wedge\ \rho \dashrightarrow^{\boldsymbol{\alpha}:\texttt{exch}(v),\boldsymbol{\alpha}:\texttt{exch}(v).\bot} \sigma \end{pmatrix}$$

$$\mathcal{G}_{\textbf{acc}}(\alpha, \beta, v, w)\ (s, \rho, t, \sigma) := \begin{pmatrix} s \xrightarrow{\boldsymbol{\beta}:\texttt{cas}(\texttt{Off}(\alpha,v),\texttt{Acc}(\alpha,\beta,v,w)).\texttt{true}} t \\ \wedge\ \rho \dashrightarrow^{\boldsymbol{\alpha}:\texttt{exch}(v),\boldsymbol{\beta}:\texttt{exch}(w),\boldsymbol{\alpha}:\texttt{exch}(v).w,\boldsymbol{\beta}:\texttt{exch}(w).v} \sigma \end{pmatrix}$$

$$\mathcal{G}[\alpha] \triangleq \mathcal{G}_{\textbf{fail}}(\alpha, -) \cup \mathcal{G}_{\textbf{acc}}(\alpha, -, -, -) \cup \mathcal{G}_{\textbf{off}}(\alpha, -) \cup \mathcal{G}_{\textbf{fin}}(\alpha)$$

$$\mathcal{R}[\alpha] \triangleq \cup_{\alpha' \neq \alpha} \mathcal{G}[\alpha']$$

Fig. 6 shows how these relations move the proof state among the three disjuncts of the invariant. After the current thread makes the offer, other threads can only perform the

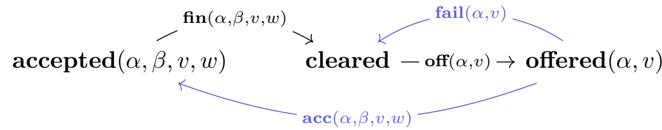


Figure 6 Logical transitions used in the proof: each node represents the assertion satisfied by the proof state, and each edge is labelled with the subscript of the guarantee condition.

accept transition $\mathcal{G}_{\textbf{acc}}$, so the precondition in the Hoare triple is stable. The post-condition is also stable because no guarantee condition can step outside the invariant.

Other commands are verified similarly, and together, they establish the entire Hoare triple for the `exch` method:

$$\mathcal{R}[\alpha], \mathcal{G}[\alpha] \vdash \{I; \texttt{Inv}[\alpha]()\}\, M[\alpha]^{\texttt{exch}}\, \{I\}$$

which discharges the remaining obligation in the module judgment of the exchanger. We refer readers to the extended version and the Rocq mechanization for more details.

5.6 Further Examples

Other than the elimination-backoff stack, we also verified (1) a family of lock-protected objects, proving correct the pattern of protecting operations of any racy but atomic objects with locks, and (2) the one-shot write-snapshot object [5], which is an interval linearizable object that no existing system can verify.

5.6.1 Lock-Protected Objects

One of the most common approaches to lift a sequential implementation of an object to an atomic linearizable concurrent object is a lock-protected critical region. Sequential implementations are prone to races when running unprotected in a concurrent setting. We prove that for *any* sequential but racy object E, if we enclose each of its operations with a pair of lock operations `acq` and `rel`, the overlay object built above the lock and E is atomic linearizable.

For example, Fig. 7 shows the structure of an atomic counter built above the lock and a racy counter. The racy counter specification (Fig. 9) adds an undefined behavior state (`UB`) to the atomic version, which would be entered whenever a concurrent access takes place. While in the `UB` state, any transition back into `UB` is allowed (as long as it does not break local sequentiality). The main proof obligation in verifying the atomic counter, i.e., $((\texttt{V_Lock} \otimes \texttt{V_RCounter}) \rhd \texttt{M_Counter}) \rightsquigarrow \texttt{V_Counter}$, is to ensure that the racy counter would never enter the `UB` state. Thanks to observational refinement, the lock specification (Fig. 8) ensures the atomic counter would never observe concurrent `acq`s – we gain mutual exclusion of `acq` − `rel` pairs "for free". As racy counter accesses only occur inside the critical region, concurrent counter accesses are never observed, and thus we show the overlay counter is safe and atomic.

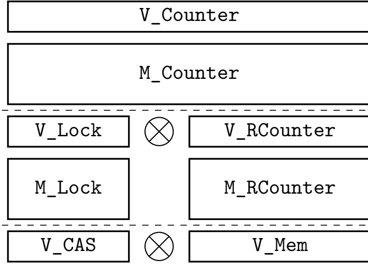


Figure 7 Module structure of the atomic counter. `RCounter` and `Counter` denote the racy counter and the atomic counter, respectively.

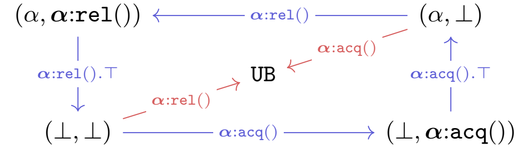


Figure 8 Lock specification.

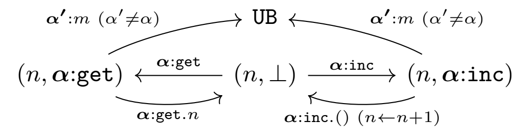


Figure 9 Racy counter specification. Atomic counter specification can be obtained by removing the `UB` state.

Instead of directly performing the above proof for the counter object, we first prove Theorem 9 holds for racy objects in general and then instantiate it with the racy counter.

▶ **Theorem 9.** *If `V_E` is an atomic LTS (invocations are immediately followed by responses of the same thread), Racy(`V_E`) is its racy version (where concurrent invocations lead to the `UB` state), and `M_E` lifts operations in Racy(`V_E`) with `acq` − `rel` pairs, then* $((\texttt{V_Lock} \otimes \mathit{Racy}(\texttt{V_E})) \rhd \texttt{M_E}) \rightsquigarrow \texttt{V_E}$.

We also verified a ticket lock implementation satisfying the lock specification in Fig. 8, which can then be connected with any racy object to provide a correct implementation of a system such as the one in Fig. 7.

$$\boldsymbol{\alpha}{:}\texttt{ws}(v)\ (\alpha{\notin}p, p{\leftarrow}p{\cup}\{\alpha\}, vs{\leftarrow}vs{\cup}\{v\}) \circlearrowright (vs,p) \circlearrowleft \boldsymbol{\beta}{:}\texttt{ws}(v).vs\ (\beta{\in}p)$$

Figure 10 Interval-sequential spec. of the write-snapshot

5.6.2 Write-Snapshot

Snapshot objects are common in distributed systems – for instance, in implementing a consensus protocol. The particular write-snapshot implementation we verify [5] is our example of an interval linearizable object, which no existing system can verify. In addition, we are able to later use such objects, integrating them as components into other systems using vertical and horizontal composition. The specification of the write-snapshot is presented in Fig. 10.

In our system, interval-sequential specs have no restriction on the ordering of events. For instance, the spec in Fig. 10 validates the trace:

$$\boldsymbol{\alpha}{:}\texttt{ws}(3) \cdot \boldsymbol{\beta}{:}\texttt{ws}(5) \cdot \boldsymbol{\alpha}{:}\texttt{ws}(3).\{3,5\} \cdot \boldsymbol{\gamma}{:}\texttt{ws}(2) \cdot \boldsymbol{\beta}{:}\texttt{ws}(5).\{3,5,2\}$$

5.7 Soundness and Completeness

We prove our program logic to be both sound (Theorem 10) and complete (Theorem 11) with regard to compositional linearizability. We outline the proofs here.

▶ **Theorem 10** (Soundness). *If* $\texttt{V_E}, \texttt{V_F}, \mathcal{R}, \mathcal{G}, P, Q \vDash M$, *then* $V_E \rhd M \rightsquigarrow V_F$.

Proof. Recall that $\texttt{V_E} \rhd M$ is a transition system which runs the code of M on top of `V_E`, i.e., there are four kinds of steps that can be made – an overlay call, an overlay return, an underlay call, or an underlay return. To show $\texttt{V_E} \rhd M \rightsquigarrow V_F$, we can show that for every transition in $\texttt{V_E} \rhd M$, there are zero or more corresponding transitions in `V_F` such that the two executions have the same overlay events. The module judgment and commit judgments directly provide the mappings from each individual step of $\texttt{V_E} \rhd M$ to `V_F`, so for the soundness proof the task is to assemble those individual steps into a full execution.

We maintain that at all steps, for each thread α, the following invariants hold for the proof state $(s, \overline{\rho})$:

1. If the thread state of α is `Idle`, then for all $f \in F$, $P[\alpha]_f(s, \overline{\rho})$ holds.
2. If the thread state of α is $\texttt{Cont}\ f\ p$ for some $f \in F$, then there exists some I such that the following holds.

$$\mathcal{R}[\alpha], \mathcal{G}[\alpha] \vdash \{P[\alpha]_f(s,\overline{\rho}); \texttt{Inv}(f); I\}\, p\, \{Q[\alpha]_f\}$$
$$\wedge\, (P[\alpha]_f(s,\overline{\rho}); \texttt{Inv}(f); I)(s,\overline{\rho}) \wedge \texttt{Stable}(\mathcal{R}[\alpha]_f, I)$$

3. If the thread state of α is $\texttt{UCall}\ f\ g\ k$ for some $f \in F$ and $g \in E$, then there should exist some I and S such that the following holds.

$$\mathcal{R}[\alpha], \mathcal{G}[\alpha] \vdash \{\mathcal{G}[\alpha]_f \vdash_\alpha \{P[\alpha]_f(s,\overline{\rho}); \texttt{Inv}(f); I\}\, g\, \{S\}$$
$$\wedge\, P[\alpha]_f(s,\overline{\rho}); \texttt{Inv}(f); I; S\}\, p\, \{Q[\alpha]_f\} \wedge \texttt{Stable}(\mathcal{R}[\alpha]_f, I)$$
$$\wedge\, \texttt{Stable}(\mathcal{R}[\alpha]_f, S) \wedge (P[\alpha]_f(s,\overline{\rho}); \texttt{Inv}(f); I)(s,\overline{\rho})$$

The proof then steps through the execution of $\texttt{V_E} \rhd M$, cycling through the invariants in order for each thread.

707 For the overlay steps, we appeal to the precondition and postcondition of the program
708 judgment. The module judgment requires the user to verify the operations with precondition
709 $P[\alpha]_f; \texttt{Inv}(f)$ and postcondition $Q[\alpha]_f$. $\texttt{Inv}(f)$ is defined to make a simultaneous step in
710 the operational semantics and the linearization, which takes care of that proof obligation.
711 Finally, the module judgment has a side condition which asks the user to prove that a
712 simultaneous overlay return and linearization return may be made if $Q[\alpha]_f$ holds of the state.
713 This side-condition is present in the mechanization but omitted here, as it is almost always
714 trivial but complicates the presentation.
715 For the underlay steps, recall that the commit judgment requires the user to directly
716 provide linearization steps for the underlay event. Thus for this case we update I or supply
717 S, and then simply add the provided steps to the linearization.
718 Once we assemble linearizations for the execution of each operation, we can chain those
719 linearizations together to form a full linearization for the operational semantics of the entire
720 concurrent object, proving $V_E \rhd M \rightsquigarrow V_F$. ◀

▶ **Theorem 11** (Completeness). *If $V_E \rhd M \rightsquigarrow V_F$, then*

$$\exists \mathcal{R}\, \mathcal{G}\, P\, Q.\ \texttt{V_E}, \texttt{V_F}, \mathcal{R}, \mathcal{G}, P, Q \vDash M$$

721 **Proof.** For completeness, we note that the possibility update rules in Fig. 4 may be extended
722 to be labelled by the same events as in Fig. 3. This extension is completely determined by
723 the particular instantiation of the update rules. For example, if in Invoke, we have that ρ_C
724 goes to $\rho_C[\alpha \mapsto m]$, then the corresponding event is the overlay invocation $\boldsymbol{\alpha}{:}m$. As a result,
725 given a sequence of possibility updates q, we may take its projection $q\restriction_{-,F}$ to the overlay
726 events implied by the rule applications (Invoke and Return rules), or its projection $q\restriction_{E,-}$
727 to only underlay events (CCommit or RCommit).
728 Completeness requires constructing $\mathcal{R}$, $\mathcal{G}$, P, and Q such that the program logic judgment
729 holds. To do this, we define an invariant $I(s, \overline{\rho})$, which maintains that there exists a trace p
730 (of the operational semantics in Fig. 3) such that the following three propositions hold.
731 **1.** The current state s is reachable by transitioning from the initial state to s along p
732 following the operational semantics.
733 **2.** For every possibility ρ in $\overline{\rho}$, ρ is reachable through possibility updates q with the same
734 overlay events as p (i.e., $q\restriction_{-,F} = p\restriction_{-,F}$).
735 **3.** For every possibility σ reachable by transitioning along the linearized specification on a
736 trace q with the same overlay events as p (i.e., $q\restriction_{-,F} = p\restriction_{-,F}$), there is a prefix l of q
737 such that a possibility ρ has transitioned along l, and there is a trace r such that ρ has
738 transitioned along r into σ.

Informally, I maintains that the set of possibilities at any point contains exactly all possibilities that are still valid w.r.t. the current concrete trace, and that are reachable in the linearized specification. The rely-guarantee conditions are then defined using I as follows:

$$P(s, \overline{\rho}) \iff I(s, \overline{\rho}) \qquad Q(_, _, t, \overline{\sigma}) \iff I(t, \overline{\sigma}) \qquad \mathcal{R}(s, \overline{\rho}, t, \overline{\sigma}) \iff I(s, \overline{\rho}) \Rightarrow I(t, \overline{\sigma})$$

$$\mathcal{G}(_, _, t, \overline{\sigma}) \iff I(t, \overline{\sigma})$$

739 From here we must prove the actual program logic judgments. The side-conditions on I,
740 such as stability, are straightforward due to how I is defined. The key obligation of the
741 program judgment is then to show that steps maintain the invariant. To ensure this, upon
742 each commit, we advance the possibility set as far as possible, populating it with every
743 possible linearization of the current execution that satisfies the invariant. Once we reach
744 the end of the function, we perform the return step by again advancing the possibilities in

745 the set as far as possible, and then pruning all the possibilities which are not ready to make
746 the overlay return. The assumption that the program is already linearizable is key here,
747 as it allows us to prove that after the pruning is complete, the possibility set will still be
748 non-empty, allowing us to complete the overlay return step. ◀

749 # 6 Related Work

750 ## 6.1 Elimination-Backoff Stack and Write-Snapshot

751 Ours is the first paper with the technology to mechanize the proof of linearizability of the
752 one-shot write-snapshot [5] object against its interval-sequential specification, and we believe
753 we are also the first to give a truly modular proof of the EBStack. We compare other works
754 that have verified the EBStack at the end of this section.

755 ## 6.2 Possibility-Based Reasoning

756 Our approach is mainly inspired by [15], [20], and [26]. [15] not only proposed the notion of
757 linearizability but also introduced possibility reasoning as an axiomatic method for proving
758 atomic linearizability, and showed it is complete and sound for proofs of linearizability
759 for individual traces. Their original technique has been recently mechanized by [16].Their
760 technique focuses only on atomicity, however, and only on verifying a single component, and
761 does not provide the ability to verify large object systems either modularly or compositionally.
762 Our approach is strictly more general in that it handles non-atomic objects and is compatible
763 with theirs, so we could have verified any example they can verify.
764 [20] packaged possibility reasoning in a program logic for the first time, and used interval
765 partial orders to represent a set of possibilities of an execution. [25, 26] show that [20]'s
766 program logic is not complete for atomically linearizable objects, and extend it further to
767 support compositional linearizability. They conjectured that their program logic was complete
768 due to its relationship with possibility reasoning. Neither work has been mechanized. Our
769 program logic mechanization builds upon their theory, and extends the possibilities from [26]
770 with state, making possibility reasoning simpler and more usable than using either partial
771 orders or traces, and embeds this method into a mechanized program logic. We provide a
772 further extensive comparison with the closely related work of [26] in the extended version of
773 the paper [13].

774 ## 6.3 Other Works

775 It is worth mentioning that other than Herlihy-Wing possibilities, [27] have provided a
776 technique based on forward and backward simulation that is sound and complete for Herlihy-
777 Wing linearizability. Backward simulations are often deemed hard to reason about, however,
778 because they go against the natural forward flow of executions, introduce backward non-
779 determinism, require knowledge of the future, and often require introducing history variables
780 to provide information that would be available under a forward technique. A similar approach
781 is that of [6], who prove a direct simulation between the EB stack and its abstract model.
782 However, this simulation is based on the execution of the entire thread pool. When showing
783 the simulation, the prover needs to discuss the execution of all threads. In contrast, our
784 approach and other program logics focus on enabling thread-local reasoning, which is more
785 accessible to programmers and makes proofs more intuitive.
786 Regarding the mechanization, the closest work to ours is [21]. It focuses on using
787 linearizability to verify serializability. Our encoding of modules as interaction trees is the

788 same as theirs, and some aspects of their treatment of atomic linearizability are compatible
789 with ours. Despite that, their framework is restricted to atomicity. This means that their
790 LTSs pair call and return events and cannot handle non-atomic objects. As a result, they
791 can't compute overlay objects ($\mathtt{V} \rhd M$), as these can be non-atomic even if V is atomic, so
792 they define an object as a pair of an underlay (atomic) LTS and a module instead. Meanwhile,
793 an object for us is just an LTS. While they prove several compositional properties of the
794 modules, which boil down to properties of interaction trees, we prove several more properties
795 related to computing overlay objects, monotonicity w.r.t. trace refinement, etc. These
796 enable the compositional linearizability framework and make up a significant portion of our
797 mechanization. Due to the compatibility of their approach with ours, we believe that their
798 results around encoding serializability as atomic linearizability could be carried over.
799 Although previous linearizability-based works cannot support non-atomic objects like
800 the exchanger, it is still possible to use Hoare triples to give them a tight specification. [28]
801 specifies the functional correctness of the exchange operation through Hoare triples and
802 develops a Hoare logic verifying the exchanger in Rocq. However, due to the limitation of
803 functional correctness, their Hoare triple specification can only be used when the execution
804 of all interleaving threads is known, i.e., composed through the parallel composition rule.
805 In contrast, our work focuses on object-level verification, where the client would access the
806 object in an arbitrary concurrent environment. Moreover, [28] only verified a simple example
807 as the client of the exchanger, instead of a complex object such as the elimination-backoff
808 stack. It is unclear whether their logic can appropriately handle complex examples such as
809 the helping mechanism in the elimination-backoff stack or future-dependent linearizations.

810 6.4 Logical Atomicity

811 Logical atomicity [7, 18] uses logically atomic Hoare triples $\langle P\rangle \mathtt{op()} \langle Q\rangle$ to specify the ability
812 of the given operation to perform the update abstracted as a transition from assertion P to
813 assertion Q at a single physically atomic statement. Since its focus is on atomicity, it cannot
814 generally specify set and interval linearizable objects satisfactorily. It is compositional as
815 one may directly use the logically atomic Hoare triple in proofs of other functions that use
816 this operation. It is known that a logically atomic triple implies Herlihy-Wing linearizability
817 [2], but there is no evidence that logical atomicity can be used to specify all Herlihy-Wing
818 linearizable objects, so its completeness is unknown. Moreover, logically atomic triples rely
819 on a concrete assertion language, a concrete programming language, and their underlying
820 semantic models. This limits the compositionality to within its own program logic framework,
821 while our approach can specify any object using an abstract LTS, without assuming the
822 actual semantic model or the program logic verifying the object against the specification.

823 6.5 Contextual Refinement

824 Contextual refinement provides an alternative to linearizability. It uses an abstract piece
825 of code A_E to specify the concrete implementation M_E by requiring that the behavior
826 of M_E running in any context C is bounded by the behavior of A_E running in the same
827 context, usually denoted as $\mathtt{M_E} \sqsupseteq \mathtt{A_E}$. Contextual refinement is equivalent [9] to Herlihy-
828 Wing linearizability and supports vertical composition in a sequential setting [29]. However,
829 contextual refinement relies on a concrete programming language for the abstract code, and
830 an embedding of this language into the original language for the concrete code is necessary
831 for composing a concrete context with an "abstract" specification. In contrast, our framework
832 only uses the signature of an object and applies to any programming language whose behaviors

are characterized by call and return events, without any modification to the language. This is a major benefit of verifying objects directly against linearizability specifications. Moreover, there has been no work to support modular verification of non-atomic objects like the exchanger and the interval-linearizable write-snapshot within the contextual refinement framework.

6.6 Relational Hoare logic

Relational Hoare logic [1] is widely used to show contextual refinement. [22] used local rely-guarantee reasoning to implement a relational logic that verifies the contextual refinement of concurrent objects w.r.t. their atomic abstract code. [34] later applied this technique to verify an operating system using their relational program logic. They achieved this using logically atomic triples [7] to establish a contextual refinement between the two. Their work allows modular verification, but with limited encapsulation: an object's abstract code `A_F` must also include all its sub-components' abstract code `A_E`. They can verify sub-components against corresponding abstract code `A_E` and use their Hoare triples to execute `A_E` embedded in `A_F`. Consequently, the top-level specification consists of very complicated abstract code that does not intuitively specify its behaviors. In contrast, our LTS specification encapsulates all details of the underlay object and provides an implementation-agnostic interface for users to verify overlay objects.

[30] encodes the contextual refinement obligation into a ghost state to verify it with CSL. Iris [18] is a higher-order logic for verifying fine-grained concurrent programs and also features logical atomicity. It leads to the ReLoC framework [10, 11, 32], which shows contextual refinement of concurrent programs using a program logic, with prophecy variables [19] enabling future-dependent linearization. ReLoC achieves compositionality by first encapsulating reusable code into smaller functions, then verifying these functions against logically atomic Hoare triples encoded as a logical relation, and lastly verifying the refinement of the top-level object using these intermediate proofs. However, it is unknown how to reuse the top-level refinement specification when verifying objects that use this top-level object. As a result, every time they build new objects using the one verified against the refinement specification, they need to prove a different functional correctness specification before they can reuse its proof. In contrast, our framework imposes LTS specifications uniformly at any level of abstraction.

As shown in Table 1, both lines of work have verified the elimination-backoff stack but failed to achieve the same level of compositionality as we do. [22] directly inlined all underlay code in the `EBStack` code. ReLoC [10] broke down the exchanger into three atomic operations: `make_offer`, `accept_offer`, and `revoke_offer`, exposing implementation details, and implemented the `EBStack` using these operations. They built and verified the `EBStack` using these three methods and their specifications. This effectively inlines a specific protocol for implementing the exchanger object within the stack implementation. As a result, to verify other objects using the exchanger, nothing except the specifications for these low-level atomic operations can be salvaged and reused from their proof. They still need to inline all the implementation code of the exchanger in the overlay object. In contrast, our approach allows direct composition above the ready-to-use exchanger implementation and specification.

7 Evaluation and Conclusion

Table 1 shows the evaluation of our program logic. There are mainly two other lines of work that mechanized the verification of the elimination-backoff stack. Neither can handle

non-atomic objects, such as the exchanger and the elimination array, compositionally, and must embed them in their stack code, while LHL allows compositional verification of all three objects.

		EBStack	Exchanger	Elimination Array	TryStack*
R-LRG[22]		◐	○	○	○
ReLoC[11]		◐	○	○	○
LHL		●	●	●	●
LOC of LHL in Rocq	**Spec**	50	38	$0^{\dagger}$	38
	Code	26	28	4	21
	Proof	4056	4244	6230	3046
	Sound	**Complete**	**Definitions & Lemmas**		
	1077	412	812		

Table 1 Evaluation of the elimination-backoff stack proof in LHL and other frameworks and lines of Rocq code of LHL. We use ○ for not verified, ◐ for verified but with sub-modules inlined, and ● for compositionally verified. The Elimination Array requires no lines of Spec because the Spec is the same as for the Exchanger. (∗) Both R-LRG and ReLoC can verify the try stack as a standalone object but they chose not to in their formalization. (†) The elimination array uses the same specification as the exchanger.

Although we have simplified possibility reasoning using LTS instead of partial orders and traces, the Rocq proof burden still turns out to be significant. As Table 1 indicates, the proofs are admittedly lengthy, reflecting the inherent complexity of reasoning about concurrent behavior. This is a consequence of a few factors. First, we did not design the program logic for ease of use, instead attempting to ensure that the proofs of soundness and completeness are easier to write. Nonetheless, a more usable program logic could be implemented and validated by translating its proofs into proofs in our more fine-grained logic. A second issue is that we have not implemented many tactics and auxiliary automation. We view proof automation as orthogonal to our contribution; our logic provides the semantic foundations upon which future automation can be built. The major difficulty we encounter in the soundness and completeness proofs is reasoning about execution traces. Even though we can hide traces from logic users using LTS, the internal proof still requires direct management of traces. We believe we can solve these issues with more automation, building on top of the foundation of LHL laid by this paper.

References

1. Nick Benton. Simple relational correctness proofs for static analyses and program transformations. *ACM SIGPLAN Notices*, 39(1):14–25, 2004. `doi:10.1145/964001.964003`.
2. Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. Theorems for free from separation logic specifications. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–29, 2021. `doi:10.1145/3473586`.
3. Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming (extended abstract). In Jim Anderson and Sam Toueg, editors, *Proceedings of the Twelth Annual ACM Symposium on Principles of Distributed Computing, Ithaca, New York, USA, August 15-18, 1993*, pages 41–51. ACM, 1993. `doi:10.1145/164051.164056`.
4. Jeremy S Bradbury, James R Cordy, and Juergen Dingel. Mutation operators for concurrent java (j2se 5.0). In *Second Workshop on Mutation Analysis (Mutation 2006-ISSRE Workshops 2006)*, pages 11–11. IEEE, 2006.
5. Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Specifying concurrent problems: Beyond linearizability and up to tasks. In *Proceedings of the 29th International Symposium*

910 *on Distributed Computing - Volume 9363*, DISC 2015, page 420–435, Berlin, Heidelberg, 2015.
911 Springer-Verlag. doi:10.1007/978-3-662-48653-5_28.

912 **6** Robert Colvin and Lindsay Groves. A scalable lock-free stack algorithm and its verification.
913 In *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM*
914 *2007)*, pages 339–348. IEEE, 2007.

915 **7** Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time
916 and data abstraction. In Richard E. Jones, editor, *ECOOP 2014 - Object-Oriented Programming*
917 *- 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume
918 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer, Springer, 2014. URL:
919 https://doi.org/10.1007/978-3-662-44202-9_9, doi:10.1007/978-3-662-44202-9_9.

920 **8** Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack.
921 In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of*
922 *Programming Languages*, POPL '15, page 233–246, New York, NY, USA, 2015. Association
923 for Computing Machinery. doi:10.1145/2676726.2676963.

924 **9** Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for
925 concurrent objects. In Giuseppe Castagna, editor, *Programming Languages and Systems,*
926 *18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European*
927 *Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29,*
928 *2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 252–266, Berlin,
929 Heidelberg, 2009. Springer. URL: https://doi.org/10.1007/978-3-642-00590-9_19, doi:
930 10.1007/978-3-642-00590-9_19.

931 **10** Dan Frumin, Robbert Krebbers, and Lars Birkedal. Reloc: A mechanised relational logic for
932 fine-grained concurrency. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd*
933 *Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July*
934 *09-12, 2018*, pages 442–451. ACM, 2018. doi:10.1145/3209108.3209174.

935 **11** Dan Frumin, Robbert Krebbers, and Lars Birkedal. Reloc reloaded: A mechanized relational
936 logic for fine-grained concurrency and logical atomicity. *Log. Methods Comput. Sci.*, 17(3),
937 2021. doi:10.46298/lmcs-17(3:9)2021.

938 **12** Éric Goubault, Jérémy Ledent, and Samuel Mimram. Concurrent specifications beyond
939 linearizability. In Jiannong Cao, Faith Ellen, Luís Rodrigues, and Bernardo Ferreira, editors,
940 *22nd International Conference on Principles of Distributed Systems, OPODIS 2018, Hong*
941 *Kong, December 17-19, 2018*, volume 125 of *LIPIcs*, pages 28:1–28:16. Schloss Dagstuhl -
942 Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.OPODIS.2018.28.

943 **13** Eashan Hatti, Arthur Oliveira Vale, Zhongye Wang, Yueyang Feng, and Zhong Shao. A
944 complete program logic for compositional linearizability. Technical Report YALEU/DCS/TR-
945 1577, Yale University, 2026. URL: https://flint.cs.yale.edu/publications/lhl.html.

946 **14** Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm.
947 In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and*
948 *architectures*, pages 206–215, 2004. doi:10.1145/1007912.1007944.

949 **15** Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent
950 objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990. doi:10.1145/78969.
951 78972.

952 **16** Prasad Jayanti, Siddhartha Jayanti, Ugur Yavuz, and Lizzie Hernandez. A universal, sound,
953 and complete forward reasoning technique for machine-verified proofs of linearizability. *Proc.*
954 *ACM Program. Lang.*, 8(POPL), jan 2024. doi:10.1145/3632924.

955 **17** Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM*
956 *Trans. Program. Lang. Syst.*, 5(4):596–619, oct 1983. doi:10.1145/69575.69577.

957 **18** Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek
958 Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation
959 logic. *Journal of Functional Programming*, 28:e20, 2018. doi:10.1017/S0956796818000151.

960 **19** Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek
961 Dreyer, and Bart Jacobs. The future is ours: prophecy variables in separation logic. *Proceedings*
962 *of the ACM on Programming Languages*, 4(POPL):1–32, 2019. doi:10.1145/3371113.
963 **20** Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew Parkinson. Proving linearizability
964 using partial orders. In *Programming Languages and Systems: 26th European Symposium*
965 *on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory*
966 *and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings 26*,
967 pages 639–667. Springer, 2017. doi:10.1007/978-3-662-54434-1_24.
968 **21** Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C.
969 Pierce, and Steve Zdancewic. C4: verified transactional objects. *Proc. ACM Program. Lang.*,
970 6(OOPSLA1), apr 2022. doi:10.1145/3527324.
971 **22** Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed lineariza-
972 tion points. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language*
973 *Design and Implementation*, pages 459–470, 2013. doi:10.1145/2491956.2462189.
974 **23** Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations, II: timing-based
975 systems. *Inf. Comput.*, 128(1):1–25, jul 1996. doi:10.1006/inco.1996.0060.
976 **24** Gil Neiger. Set-linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium*
977 *on Principles of Distributed Computing*, PODC '94, page 396, New York, NY, USA, 1994.
978 Association for Computing Machinery. doi:10.1145/197917.198176.
979 **25** Arthur Oliveira Vale, Zhong Shao, and Yixuan Chen. A compositional theory of linearizability.
980 *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi:10.1145/3571231.
981 **26** Arthur Oliveira Vale, Zhong Shao, and Yixuan Chen. A compositional theory of linearizability.
982 *J. ACM*, 71(2), apr 2024. doi:10.1145/3643668.
983 **27** Gerhard Schellhorn, John Derrick, and Heike Wehrheim. A sound and complete proof technique
984 for linearizability of concurrent data structures. *ACM Trans. Comput. Logic*, 15(4), sep 2014.
985 doi:10.1145/2629496.
986 **28** Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. Hoare-
987 style specifications as correctness conditions for non-linearizable concurrent objects. *ACM*
988 *SIGPLAN Notices*, 51(10):92–110, 2016.
989 **29** Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer.
990 Conditional contextual refinement. *Proceedings of the ACM on Programming Languages*,
991 7(POPL):1121–1151, 2023. doi:10.1145/3571232.
992 **30** Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and hoare-style reasoning
993 in a logic for higher-order concurrency. In *Proceedings of the 18th ACM SIGPLAN international*
994 *conference on Functional programming*, pages 377–390, 2013. doi:10.1145/2500365.2500600.
995 **31** Viktor Vafeiadis. *Modular fine-grained concurrency verification.* PhD thesis, University of
996 Cambridge, UK, jul 2008. URL: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.
997 612221, doi:10.48456/tr-726.
998 **32** Simon Friis Vindum, Dan Frumin, and Lars Birkedal. Mechanized verification of a fine-grained
999 concurrent queue from meta's folly library. In *Proceedings of the 11th ACM SIGPLAN*
1000 *International Conference on Certified Programs and Proofs*, pages 100–115, 2022. doi:10.
1001 1145/3497775.3503689.
1002 **33** Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce,
1003 and Steve Zdancewic. Interaction trees: representing recursive and impure programs in Coq.
1004 *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi:10.1145/3371119.
1005 **34** Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A practical
1006 verification framework for preemptive os kernels. In *International Conference on Computer*
1007 *Aided Verification*, pages 59–79. Springer, 2016. doi:10.1007/978-3-319-41540-6_4.

1008
A Verifying the Elimination Backoff Stack

1009 Once the components of the EB stack – the exchanger, elimination array, wait-free stack,
1010 and the EB stack itself – have been constructed, the modules and proofs may be composed
1011 by employing LHL's compositionality features. We verify the EB stack against a standard
1012 atomic stack specification, where pop() will fail on an empty stack. This specification is
1013 depicted in Fig. 11.

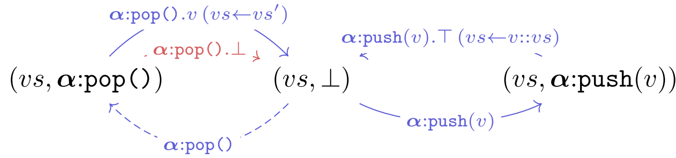


Figure 11 Stack specification – pop() returns v when $vs = v :: vs'$, and fails when $vs = []$, returning $\perp$

1014 Using vertical composition, horizontal composition, and soundness (see §5.7), we can
1015 prove the linearizability of the EB stack implementation. First, in Fig. 12 in we apply
soundness to derive linearizability proofs for each of the components.

$$\frac{\mathcal{R}, \mathcal{G} \models \{P\}\, \texttt{M_Stack}\, \{Q\}}{\otimes_{\alpha \in \mathcal{T}} \texttt{V_CAS} \rhd \texttt{M_Stack} \rightsquigarrow \texttt{V_WFStack}} \qquad \frac{\mathcal{R}, \mathcal{G} \models \{P\}\, \texttt{M_ExchArr}\, \{Q\}}{\otimes_{\alpha \in \mathcal{T}} \texttt{V_Exch} \rhd \texttt{M_ExchArr} \rightsquigarrow \texttt{V_Exch}}$$

$$\frac{\mathcal{R}, \mathcal{G} \models \{P\}\, \texttt{M_EBStack}\, \{Q\}}{\texttt{V_Exch} \otimes \texttt{V_WFStack} \rhd \texttt{M_EBStack} \rightsquigarrow \texttt{V_WFStack}}$$

Figure 12 Individual linearizability derivations for the components of the EB Stack

1016

1017 Once we have extracted proofs of linearizability from each of the LHL proofs, we proceed
1018 to iteratively applying vertical and horizontal composition in order to produce our final
1019 linearizability proof. First, we vertically compose an array of one-cell exchangers and the
1020 exchanger array (Fig. 13). Second, we horizontally compose this exchanger array and the
1021 wait-free stack (Fig. 14). Finally, we vertically compose this product with the EB stack,
1022 producing the desired proof of linearizability (Fig. 15).

$$\frac{\begin{array}{c}\otimes_{\alpha \in \mathcal{T}} \texttt{V_CAS} \rhd \texttt{M_Exch} \rightsquigarrow \texttt{V_Exch} \\ \otimes_{\alpha \in \mathcal{T}} \texttt{V_Exch} \rhd \texttt{M_ExchArr} \rightsquigarrow \texttt{V_Exch}\end{array}}{\otimes_{\alpha \in \mathcal{T}\mathcal{T}} \texttt{V_CAS} \rhd (\texttt{M_Exch} :\rhd \texttt{M_ExchArr}) \rightsquigarrow \texttt{V_Exch}}$$

Figure 13 Vertical composition of one-cell exchanger and elimination array

$$\frac{\otimes_{\alpha\in\mathcal{T}^{\mathcal{T}}}\texttt{V_CAS} \rhd (\texttt{M_Exch} :\rhd \texttt{M_ExchArr}) \leadsto \texttt{V_Exch} \qquad \otimes_{\alpha\in\mathcal{T}}\texttt{V_CAS} \rhd \texttt{M_Stack} \leadsto \texttt{V_WFStack}}{\otimes_{\alpha\in(\mathcal{T}^{\mathcal{T}}+1)}\texttt{V_CAS} \rhd \begin{pmatrix}(\texttt{M_Exch} :\rhd \texttt{M_ExchArr}) \otimes \\ \texttt{M_Stack}\end{pmatrix} \leadsto \texttt{V_WFStack}}$$

Figure 14 Horizontal composition of elimination array and wait-free stack

$$\frac{\otimes_{\alpha\in(\mathcal{T}^{\mathcal{T}}+1)}\texttt{V_CAS} \rhd ((\texttt{M_Exch} :\rhd \texttt{M_ExchArr}) \otimes \texttt{M_Stack}) \leadsto \texttt{V_WFStack} \qquad \texttt{V_Exch} \otimes \texttt{V_WFStack} \leadsto \texttt{V_WFStack}}{\otimes_{\alpha\in(\mathcal{T}^{\mathcal{T}}+1)}\texttt{V_CAS} \rhd (((\texttt{M_Exch} :\rhd \texttt{M_ExchArr}) \otimes \texttt{M_Stack}) :\rhd \texttt{M_EBStack}) \leadsto \texttt{V_WFStack}}$$

Figure 15 Final proof of linearizability for the EB Stack

1023 ## A.1 Wait-Free Stack

1024 Fig. 16 defines the LTS of the memory object. We annotate it with a partial map *loc* from
1025 locations to the thread that allocates the location. This information is necessary for us to
1026 ensure each thread only writes to the memory cell allocated by itself in the overlay proof.
1027 Fig. 17 defines the LTS of the wait-free stack.
1028 Assume the CAS cell's initial state is $(\bot, \bot)$, where the first $\bot$ denotes an empty address
1029 and the second denotes there are no pending operations.

Define `listseg` to be the minimal predicate satisfying the following condition.

$$\texttt{listseg}(h, l_1, l_2, vs) \iff \begin{array}{c}(l_1 = l_2 \wedge vs = []) \vee \\ \begin{pmatrix} h(l_1) = (v, l')\wedge \\ vs = v :: vs'\wedge \\ \texttt{listseg}(h, l', l_2, vs')\end{pmatrix}\end{array}$$

The following invariant holds throughout the execution. It asserst that (1) the list segment represented by the underlay LTS matches the stack in the overlay LTS, and (2) a thread can

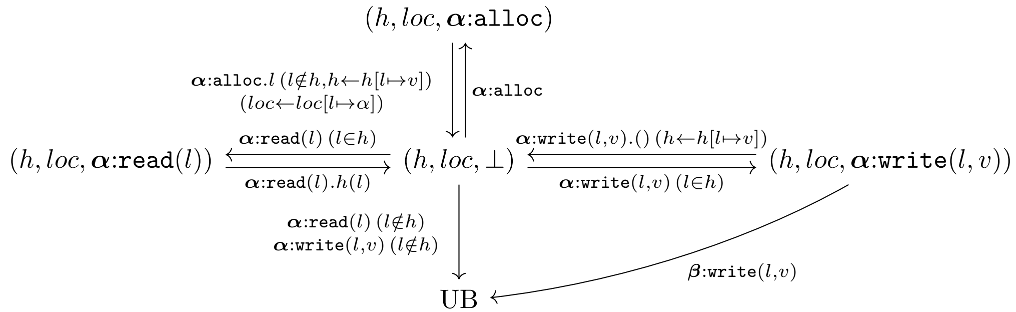


Figure 16 Memory Specification: initial state is $(\{\}, \bot)$. *loc* is only an auxilliary state for recording the thread that allocates each location. The auxilliary state does not affect transitions, and we can show the refinement between it with the one without the auxilliary state.

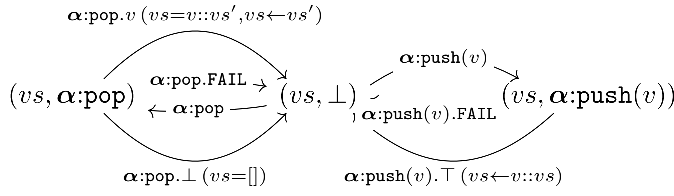


Figure 17 Wait-Free Stack Specification: initial state is $([], \bot)$

only write to a memory cell that is allocated by itself.

$$I(s,\rho) := \begin{pmatrix} \begin{pmatrix} \rho = (vs, \bot) \wedge \\ s_{\texttt{cas}} = (l, -) \wedge s_{\texttt{mem}} = (h, -, -) \wedge \\ \texttt{listseg}(h, l, \bot, vs) \end{pmatrix} \wedge \\ (\forall \alpha.\, s_{\texttt{mem}} = (-, loc, \boldsymbol{\alpha}{:}\texttt{write}(l, -)) \Rightarrow loc(l) = \alpha) \end{pmatrix}$$

We then have the following guarantee relations defining the abstracted concrete transition system. Notice that these relations excludes transitions that lead to UB state.

$$(s,\rho)\ \mathcal{G}_{\text{empty}}[\alpha]\ (t,\sigma) \iff \begin{pmatrix} s \xrightarrow{\boldsymbol{\alpha}:C.\texttt{C.get}.\bot} t \wedge \\ \rho \overset{\boldsymbol{\alpha}:\texttt{pop}\cdot\boldsymbol{\alpha}:\texttt{pop}.\bot}{\dashrightarrow} \sigma \end{pmatrix}$$

$$(s,\rho)\ \mathcal{G}_{\text{write}}[\alpha]\ (t,\sigma) \iff \begin{pmatrix} s_{\texttt{mem}} = (-, loc, -) \wedge \\ loc(l) = \alpha \wedge \\ s \xrightarrow{\boldsymbol{\alpha}:M.\texttt{write}(l,v)} t \wedge \sigma = \rho \end{pmatrix}$$

$$(s,\rho)\ \mathcal{G}_{\text{cas_push}}[\alpha]\ (t,\sigma) \iff \begin{pmatrix} s_{\texttt{mem}} = (h, -, -) \wedge \\ h(new) = (v, old) \wedge \\ s \xrightarrow{\boldsymbol{\alpha}:C.\texttt{CAS}(old,new).\texttt{true}} t \\ \wedge \rho \overset{\boldsymbol{\alpha}:\texttt{push(v)}\cdot\boldsymbol{\alpha}:\texttt{push(v)}.\top}{\dashrightarrow} \sigma \end{pmatrix}$$

$$(s,\rho)\ \mathcal{G}_{\text{cas_pop}}[\alpha]\ (t,\sigma) \iff \begin{pmatrix} s_{\texttt{mem}} = (h,-,-)\wedge \\ h(old) = (v, new)\wedge \\ s \xrightarrow{\boldsymbol{\alpha}:C.\texttt{CAS}(old,new).\texttt{true}} t\wedge \\ \rho \overset{\boldsymbol{\alpha}:\texttt{pop}\cdot\boldsymbol{\alpha}:\texttt{pop}.v}{\dashrightarrow} \sigma \end{pmatrix}$$

$$(s,\rho)\ \mathcal{G}_{\text{cas_fail}}[\alpha]\ (t,\sigma) \iff \begin{pmatrix} s \xrightarrow{\boldsymbol{\alpha}:C.\texttt{CAS}(old,new).\texttt{false}} t\wedge \\ \begin{pmatrix} \rho \overset{\boldsymbol{\alpha}:\texttt{push(v)}\cdot\boldsymbol{\alpha}:\texttt{push(v)}.\bot}{\dashrightarrow} \sigma \\ \vee\rho \overset{\boldsymbol{\alpha}:\texttt{pop}\cdot\boldsymbol{\alpha}:\texttt{pop}.\bot}{\dashrightarrow} \sigma \end{pmatrix} \end{pmatrix}$$

$$(s,\rho)\ \mathcal{G}_{\text{other}}[\alpha]\ (t,\sigma) \iff \begin{pmatrix} s \xrightarrow{m} t \wedge \rho = \sigma\wedge \\ m \text{ is some other move} \end{pmatrix}$$

$$\mathcal{G}[\alpha] \triangleq \begin{pmatrix} \mathcal{G}_{empty} \cup \mathcal{G}_{write} \cup \mathcal{G}_{cas_push}\cup \\ \mathcal{G}_{cas_pop} \cup \mathcal{G}_{cas_fail} \cup \mathcal{G}_{other}\cup \\ \texttt{Invoke}_\alpha \cup \texttt{Return}_\alpha \end{pmatrix}$$

$$\mathcal{R}[\alpha] \triangleq \bigcup_{\alpha' \neq \alpha} \mathcal{G}[\alpha']$$

Fig. 18 and Fig. 19 shows the proof outline for the push and pop operation. The proof is mostly simple, where we maintain the main invariant and ensure the preconditions for performing the overlay possibility update. We also only use singleton possibility in this proof. It is worth mentioning that after making a `write` invocation step in the `push` operation, we can ensure that this step won't lead to UB state as the rely condition ensures no other thread can modify the location allocated by the current thread.

A.2 Exchanger

We chose to implement the exchanger with a single CAS cell, the LTS of which is given in Fig. 2a. Fig. 2b defines the LTS of the exchanger itself. The CAS cell's state is denoted by $s_{\texttt{CAS}}$ and may be in one of three states:

1. $s_{\texttt{CAS}} = \bot$: No thread is currently making an offer.
2. $s_{\texttt{CAS}} = \texttt{Off}(\alpha, v)$: Thread α is currently offering the value v, but no other thread has accepted this offer yet.
3. $s_{\texttt{CAS}} = \texttt{Acc}(\alpha, \beta, v, w)$: Thread α offered v, and thread β accepted the offer, offering w in return.

The execution of the exchanger will transition on these states in order, with the addition of the following two transitions:

- If the state is $\texttt{Off}(\alpha, v)$ to $\bot$ but the offer is not accepted in time, α will transition the state back to $\bot$ and indicate failure to the client of the exchanger.
- Once the state reaches $\texttt{Acc}(\alpha, \beta, v, w)$, α will transition it back to $\bot$. The exchange was successful, so α returns w and β returns v.

$\{I \wedge p_C[\alpha] = \texttt{push}(v)\}$
$\texttt{push}(v)\{$
  $old \leftarrow C.\texttt{C.get}();$
  $\left\{\begin{array}{c} I \wedge p_C[\alpha] = \texttt{push} \wedge s_{\texttt{mem}} = (h, -, -)\wedge \\ \exists vs.\,\texttt{listseg}(h, old, \bot, vs) \end{array}\right\}$
  $new \leftarrow M.\texttt{alloc}();$
  $\left\{\begin{array}{c} I \wedge p_C[\alpha] = \texttt{push} \wedge s_{\texttt{mem}} = (h, loc, -)\wedge \\ loc[new] = \alpha \wedge \exists vs.\,\texttt{listseg}(h, old, \bot, vs) \end{array}\right\}$
  $M.\texttt{write}(new, (v, old));$ // invoke step
  $\left\{\begin{array}{c} I \wedge p_C[\alpha] = \texttt{push} \wedge s_{\texttt{mem}} = (h, loc, -)\wedge \\ loc[new] = \alpha \wedge \exists vs.\,\texttt{listseg}(h, old, \bot, vs) \end{array}\right\}$
  $M.\texttt{write}(new, (v, old)).();$ // return step
  $\left\{\begin{array}{c} I \wedge p_C[\alpha] = \texttt{push} \wedge s_{\texttt{mem}} = (h, loc, -)\wedge \\ loc[new] = \alpha \wedge \exists vs.\,\texttt{listseg}(h, new, \bot, v :: vs) \end{array}\right\}$
  $\texttt{if}(C.\texttt{CAS}(old, new))\{$ // commit: push/fail
    $\{I \wedge p_C[\alpha] = \texttt{push} \wedge p_R[\alpha] = \top\}$
    $\texttt{ret}\ \top;$
  $\}\texttt{else}\{$
    $\{I \wedge p_C[\alpha] = \texttt{push} \wedge p_R[\alpha] = \texttt{FAIL}\}$
    $\texttt{ret}\ \texttt{FAIL};$
  $\}$
$\}$

Figure 18 Wait-Free stack proof outline: `push`

$$
\begin{array}{l}
\{I \wedge p_C[\alpha] = \texttt{pop}\} \\
\texttt{pop}()\{ \\
\quad old \leftarrow C.\texttt{C.get}(); \\
\quad \left\{ \begin{array}{l} I \wedge p_C[\alpha] = \texttt{pop} \wedge s_{\texttt{mem}} = (h, -, -) \wedge \\ \qquad \exists vs.\, \texttt{listseg}(h, old, \bot, vs) \end{array} \right\} \\
\quad \texttt{if}(old = \bot)\{ \text{ // commit: empty} \\
\qquad \left\{ \begin{array}{c} I \wedge p_C[\alpha] = \texttt{pop} \wedge p_R[\alpha] = \bot \wedge \\ s_{\texttt{mem}} = (h, -, -) \wedge \exists vs.\, \texttt{listseg}(h, \bot, \bot, []) \end{array} \right\} \\
\qquad \texttt{ret}\, \bot; \\
\quad \}\texttt{else}\{ \\
\qquad \left\{ \begin{array}{l} I \wedge p_C[\alpha] = \texttt{pop} \wedge s_{\texttt{mem}} = (h, -, -) \wedge \\ \quad \exists v, vs.\, \texttt{listseg}(h, old, \bot, v :: vs) \end{array} \right\} \\
\qquad head \leftarrow M.\texttt{read}(old); \\
\qquad \left\{ \begin{array}{l} \qquad I \wedge p_C[\alpha] = \texttt{pop} \wedge s_{\texttt{mem}} = (h, -, -) \wedge \\ \exists v, vs. \left(\begin{array}{c} h(old) = head \wedge head = (v, new) \wedge \\ \texttt{listseg}(h, new, \bot, vs) \end{array} \right) \end{array} \right\} \\
\qquad b \leftarrow C.\texttt{CAS}(old, \texttt{snd}(head)); \text{ // commit: pop/fail} \\
\qquad \texttt{if}(b)\{ \\
\qquad\quad \left\{ \begin{array}{l} I \wedge p_C[\alpha] = \texttt{pop} \wedge p_R[\alpha] = v \wedge s_{\texttt{mem}} = (h, -, -) \wedge \\ \quad \exists v, vs. \left(\begin{array}{c} h(old) = head \wedge head = (v, new) \wedge \\ \texttt{listseg}(h, new, \bot, vs) \end{array} \right) \end{array} \right\} \\
\qquad\quad \texttt{ret}\, \texttt{fst}(head); \\
\qquad \}\texttt{else}\{ \\
\qquad\quad \{I \wedge p_C[\alpha] = \texttt{pop} \wedge p_R[\alpha] = \texttt{FAIL}\} \\
\qquad\quad \texttt{ret}\, \texttt{FAIL}; \\
\qquad \} \\
\quad \} \\
\}
\end{array}
$$

Figure 19 Wait-Free stack proof outline: `pop`

$$
\begin{aligned}
(s,\rho)\mathcal{G}_{\texttt{offer}(v)}[\alpha](t,\sigma) &\iff \rho = \sigma \wedge s \xrightarrow{\boldsymbol{\alpha}:\texttt{C.cas}(\bot,\texttt{Off}(\alpha,v))} t \\
(s,\rho)\mathcal{G}_{\texttt{accept}(\beta,v,w)}[\alpha](t,\sigma) &\iff \left(\begin{array}{c} \rho \overset{\boldsymbol{\alpha}:\texttt{exch}(v),\boldsymbol{\beta}:\texttt{exch}(w),\boldsymbol{\alpha}:\texttt{exch}(v).w,\boldsymbol{\beta}:\texttt{exch}(w).v}{\dashrightarrow} \sigma \\ \wedge\, s \xrightarrow{\boldsymbol{\alpha}:\texttt{C.cas}(\texttt{Off}(\alpha,v),\texttt{Acc}(\alpha,\beta,v,w)).\texttt{true}} t \end{array}\right) \\
(s,\rho)\mathcal{G}_{\texttt{fail}(v)}[\alpha](t,\sigma) &\iff \rho \overset{\boldsymbol{\alpha}:\texttt{exch}(v),\boldsymbol{\alpha}:\texttt{exch}(v).\bot}{\dashrightarrow} \sigma \wedge s \xrightarrow{\boldsymbol{\alpha}:\texttt{C.cas}(a,\bot).\texttt{true}} t \\
(s,\rho)\mathcal{G}_{\texttt{finish}(\beta,v,w)}[\alpha](t,\sigma) &\iff \rho = \sigma \wedge s \xrightarrow{\boldsymbol{\alpha}:\texttt{C.cas}(\texttt{Acc}(\alpha,\beta,v,w),\bot).\texttt{true}} t
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{G}[\alpha] &\triangleq \left(\begin{array}{c} \mathcal{G}_{offer}[\alpha] \cup \mathcal{G}_{accept}[\alpha] \cup \mathcal{G}_{fail}[\alpha] \cup \mathcal{G}_{finish}[\alpha] \cup \\ \texttt{Invoke}_\alpha \cup \texttt{Return}_\alpha \end{array}\right) \\
\mathcal{R}[\alpha] &\triangleq \bigcup_{\alpha' \neq \alpha} \mathcal{G}[\alpha']
\end{aligned}
$$

Figure 20 Rely and guarantee definitions for the exchanger

1069 With this in mind, we define the guarantee and rely in Fig. 20.

1070 The proof outline for the `exchange`(v) operation is given in Fig. 21. Only a single
1071 possibility is necessary for the proof, so all possibility sets are implicitly the singleton set $\{\rho\}$
1072 of a possibility ρ. The assertions are simple, each maintaining the invariant alongside listing
1073 the possible states (out of the three listed above) that the CAS cell may be in. Additionally,
1074 the invariant maintains that the exchanger is always in the idle state $(\{\}, \{\})$, this is possible
1075 because the proof always performs all the steps of an exchanger interaction in a single commit.

1076 Stability of the assertions is also simple. The states listed in an assertion are merely a
1077 starting state unioned with all possible states reachable from it in the guarantee transition
1078 system. Since the rely is defined as the reflexive-transitive closure of the guarantee, stability
1079 is evident from simple induction.

$$I(s,\rho) \iff \rho = \bot$$

$$\texttt{AnyState}(s,\rho) \iff$$

$$\left(\begin{array}{c} s_{\texttt{exch}} = \bot \vee \\ \left(\exists \beta\, v, \left(\begin{array}{c}\beta \neq \alpha \wedge \\ s_{\texttt{exch}} = \texttt{Off}(\beta, v)\end{array}\right)\right) \vee \\ \exists \beta\gamma v w, \left(\begin{array}{c}\beta \neq \alpha \wedge \beta \neq \gamma \wedge \\ s_{\texttt{exch}} = \texttt{Acc}(\beta,\gamma,v,w)\end{array}\right)\end{array}\right)$$

$$\texttt{SelfOffered}(v,s,\rho) \iff$$

$$\left(\begin{array}{c} s_{\texttt{exch}} = \texttt{Off}(\alpha, v) \vee \\ \exists \beta\, w, \left(\begin{array}{c}\beta \neq \alpha \wedge \\ s_{\texttt{exch}} = \texttt{Acc}(\alpha,\beta,v,w)\end{array}\right)\end{array}\right)$$

$$\texttt{OtherAccepted}(v,s,\rho) \iff$$

$$\exists \beta\, w, \left(\begin{array}{c}\beta \neq \alpha \wedge \\ s_{\texttt{exch}} = \texttt{Acc}(\alpha,\beta,v,w)\end{array}\right)$$

```
{I ∧ ρ_C[α] = exch(v) ∧ AnyState}
exchange(v){
  offerplaced ← C.cas(⊥, Off(α, v))
  { I ∧ ρ_C[α] = exch(v) ∧
    ((offerplaced = true ∧ SelfOffered(v)) ∨
     (offerplaced = false ∧ AnyState)) }
  if(offerplaced){
    // See Fig. 22
  }else{
    // See Fig. 23
  }
}
```

Figure 21 Exchanger proof – main body

A.3 Elimination-Backoff Stack

We implement the EB stack with an exchanger and a wait-free stack. The LTS of the former is given in Fig. 2, and the latter is given in Fig. 17. Finally, the LTS of the EB stack itself – an atomic stack specification – is given in Fig.11.

Both the `push` and `pop` operations follow the same pattern: First the corresponding operation in the underlying wait-free stack is performed. If the operation is successful, no more work is necessary and the operations. Otherwise, the operations performs an exchange in hopes that the opposite operation – i.e `push` for `pop` and `pop` for `push` – is also doing so. If the exchange is successful, the underlying stack will not change and both threads will return, having cancelled out each other's operations. Otherwise, the EB stack will loop and attempt the operation again.

$\{I \wedge \rho_C[\alpha] = \texttt{exch}(v) \wedge \texttt{SelfOffered}(v)\}$
$\texttt{nochange} \leftarrow \texttt{C.cas}(\texttt{Off}(\alpha, v), \bot)$

$$\left\{ \begin{array}{c} I \wedge \rho_C[\alpha] = \texttt{exch}(v) \wedge \\ \left(\begin{pmatrix} \texttt{nochange} = \texttt{true} \wedge \texttt{SelfOffered}(v) \wedge \\ \rho_R[\alpha] = \texttt{exch}(v).\bot \end{pmatrix} \vee \\ (\texttt{nochange} = \texttt{false} \wedge \texttt{OtherAccepted}(v)) \right) \end{array} \right\}$$

$\texttt{if}(\texttt{nochange})\{$

$$\left\{ \begin{array}{c} I \wedge \rho_C[\alpha] = \texttt{exch}(v) \wedge \texttt{SelfOffered}(v) \\ \wedge \rho_R[\alpha] = \texttt{exch}(v).\bot \end{array} \right\}$$

$\texttt{ret}\ \bot$
$\}\texttt{else}\{$
$\{I \wedge \rho_C[\alpha] = \texttt{exch}(v) \wedge \texttt{OtherAccepted}(v)\}$
$r \leftarrow \texttt{C.get}()$
$\texttt{match}(r)\{$
$\texttt{Acc}(_, _, _, w) \Rightarrow \{$
$\{I \wedge \rho_C[\alpha] = \texttt{exch}(v) \wedge \texttt{OtherAccepted}(v)\}$
$\texttt{C.cas}(r, \bot)$

$$\left\{ \begin{array}{c} I \wedge \rho_C[\alpha] = \texttt{exch}(v) \wedge \texttt{AnyState} \wedge \\ \rho_R[\alpha] = \texttt{exch}(v).w \end{array} \right\}$$

$\texttt{ret}\ w$
$\}$
$_ \Rightarrow \{$
$\{\texttt{Acc}(_, _, _, _) = \texttt{Off}(_, _) \vee \texttt{Acc}(_, _, _, _) = \bot\}$
// impossible case
$\texttt{ret}\ \bot$
$\}$
$\}$
$\}$

Figure 22 Exchanger proof – `true` case for `offerplaced`

We denote the wait-free stack state as $s_{\texttt{stk}}$ and the exchanger state as $s_{\texttt{exch}}$. The rely and guarantee are defined in Fig. 24.

The high-level structure of the proof is similar to the exchanger. Only a single possibility is needed and assertions simply list the possible states the exchanger and wait-free stack are in. Additionally the invariant maintains that the overlay state is always idle, again because all possibility steps are always performed in single commits. For the same reason as in the exchanger, stability proofs are evident from simple induction on the transition system.

Proof outlines for `push` and `pop` are given in Fig. 25 and Fig. 26. Their structure is quite similar, as they both wrap around the underlying stack operation in the same manner. The loop invariant and assertions are defined below. The loop invariant maintains that the wait-free stack and EB stack are consistent with each other.

The key difficulty of the proof is that when an exchanging thread makes an offer, a previous *accepted* offer the thread made may not yet have be cleared from the exchanger state by the accepting thread. This is the *only* case in which a thread may attempt an exchange while it is technically still participating in another, and this fact is what the invariant must maintain. In other words, the invariant maintains that a thread may not offer to two threads

```
{I ∧ ρ_C[α] = exch(v) ∧ AnyState}
r ← C.get()
match(r){
  Off(β, w) ⇒ {
    {I ∧ ρ_C[α] = exch(v) ∧ AnyState}
    accepted ← C.cas(Off(β, w), Acc(β, α, w, v))
    { I ∧ ρ_C[α] = exch(v)∧
      (((accepted = true ∧ AnyState∧
         ρ_C[α] = exch(v).w) ∨
        (accepted = false ∧ AnyState∧
         ρ_C[α] = exch(v).⊥))) }
    if(accepted){
      { I ∧ ρ_C[α] = exch(v) ∧ AnyState∧
        ρ_C[α] = exch(v).w }
      ret w
    }else{
      { I ∧ ρ_C[α] = exch(v) ∧ AnyState∧
        ρ_C[α] = exch(v).⊥ }
      ret ⊥
    }
  }
  _ ⇒ {
    { I ∧ ρ_C[α] = exch(v) ∧ AnyState∧
      ρ_R[α] = exch(v).⊥ }
    ret ⊥
  }
}
```

Figure 23 Exchanger proof – `false` case for `offerplaced`

at the same time, which is evident from the exchanger LTS.

$$I(s, \rho) \iff \begin{pmatrix} s_{\mathtt{stk}} = \rho \wedge \\ \forall o\, c.\, s_{\mathtt{exch}} = (o, c) \wedge (\exists v.\, (\alpha, v) \in o) \Rightarrow c \neq \emptyset \end{pmatrix}$$

Since the `exch` method is nonatomic, we must verify it in two distinct steps. After the call (notated `.inv` in the proof), the exchanger will be in one of three states:

- **Waiting**: The exchanger is still waiting for its offer to be accepted by another thread (or to fail).
- **Completed**: Either `push` offered and `pop` accepted, or `pop` offered and `push` accepted. The elimination was successful and the operations may now return.
- **Conflict**: Either two `push`es or two `pop`s exchanged. The elimination failed and the operations must loop again.

$$(s,\rho)\mathcal{G}_{\texttt{callpush}(v)}[\alpha](t,\sigma) \iff \rho = \sigma \land s \xrightarrow{\boldsymbol{\alpha}:\text{push}(v)} t$$

$$(s,\rho)\mathcal{G}_{\texttt{pushpass}(v)}[\alpha](t,\sigma) \iff \rho \xrightarrow{\boldsymbol{\alpha}:\text{push}(v),\boldsymbol{\alpha}:\text{push}(v).\top} \sigma \land s \xrightarrow{\boldsymbol{\alpha}:\text{push}(v).\top} t$$

$$(s,\rho)\mathcal{G}_{\texttt{pushfail}(v)}[\alpha](t,\sigma) \iff \rho = \sigma \land s \xrightarrow{\boldsymbol{\alpha}:\text{push}(v).\bot} t$$

$$(s,\rho)\mathcal{G}_{\texttt{callpop}()}[\alpha](t,\sigma) \iff \rho = \sigma \land s \xrightarrow{\boldsymbol{\alpha}:\text{pop}()} t$$

$$(s,\rho)\mathcal{G}_{\texttt{poppass}()}[\alpha](t,\sigma) \iff \rho \xrightarrow{\boldsymbol{\alpha}:\text{pop}(),\boldsymbol{\alpha}:\text{pop}().\top} \sigma \land s \xrightarrow{\boldsymbol{\alpha}:\text{pop}().\top} t$$

$$(s,\rho)\mathcal{G}_{\texttt{popfail}()}[\alpha](t,\sigma) \iff \rho = \sigma \land s \xrightarrow{\boldsymbol{\alpha}:\text{pop}().\bot} t$$

$$(s,\rho)\mathcal{G}_{\texttt{offer(v)}()}[\alpha](t,\sigma) \iff \rho = \sigma \land s \xrightarrow{\boldsymbol{\alpha}:\text{exch}(v)} t$$

$$(s,\rho)\mathcal{G}_{\texttt{revoke(v)}()}[\alpha](t,\sigma) \iff \rho = \sigma \land s \xrightarrow{\boldsymbol{\alpha}:\text{exch}(v).\bot} t$$

$$(s,\rho)\mathcal{G}_{\texttt{pushaccept}(\alpha,\beta,v)}[\alpha](t,\sigma) \iff \begin{pmatrix} s_{\texttt{exch}} = (\{(\alpha,v),(\beta,\top)\},\{\})\land \\ \rho \xrightarrow{\boldsymbol{\alpha}:\text{push}(v),\boldsymbol{\alpha}:\text{push}(v).\top,\boldsymbol{\beta}:\text{pop}(),\boldsymbol{\beta}:\text{pop}().v} \sigma \land s \xrightarrow{\boldsymbol{\alpha}:\text{exch}(v).\top} t \end{pmatrix}$$

$$(s,\rho)\mathcal{G}_{\texttt{pushconflict}(\alpha,\beta,v,w)}[\alpha](t,\sigma) \iff \begin{pmatrix} s_{\texttt{exch}} = (\{(\alpha,v),(\beta,w)\},\{\})\land \\ \rho = \sigma \land s \xrightarrow{\boldsymbol{\alpha}:\text{exch}(v).w} t \end{pmatrix}$$

$$(s,\rho)\mathcal{G}_{\texttt{popaccept}(\alpha,\beta,v)}[\alpha](t,\sigma) \iff \begin{pmatrix} s_{\texttt{exch}} = (\{(\alpha,\top),(\beta,v)\},\{\})\land \\ \rho \xrightarrow{\boldsymbol{\alpha}:\text{push}(v),\boldsymbol{\alpha}:\text{push}(v).\top,\boldsymbol{\beta}:\text{pop}(),\boldsymbol{\beta}:\text{pop}().v} \sigma \land s \xrightarrow{\boldsymbol{\alpha}:\text{exch}(\top).v} t \end{pmatrix}$$

$$(s,\rho)\mathcal{G}_{\texttt{popconflict}(\alpha,\beta)}[\alpha](t,\sigma) \iff \begin{pmatrix} s_{\texttt{exch}} = (\{(\alpha,\top),(\beta,\top)\},\{\})\land \\ \rho = \sigma \land s \xrightarrow{\boldsymbol{\alpha}:\text{exch}(\top).\top} t \end{pmatrix}$$

$$(s,\rho)\mathcal{G}_{\texttt{finish}(\alpha,\beta,v,w)}[\alpha](t,\sigma) \iff \begin{pmatrix} s_{\texttt{exch}} = (\{(\beta,w)\},\{(\alpha,v)\})\land \\ \rho = \sigma \land s \xrightarrow{\boldsymbol{\alpha}:\text{exch}(v).w} t \end{pmatrix}$$

$$\mathcal{G}[\alpha] \triangleq \begin{pmatrix} \mathcal{G}_{callpush} \cup \mathcal{G}_{pushpass} \cup \mathcal{G}_{pushfail} \cup \\ \mathcal{G}_{callpop} \cup \mathcal{G}_{poppass} \cup \mathcal{G}_{popfail} \cup \\ \mathcal{G}_{offer} \cup \mathcal{G}_{revoke} \cup \mathcal{G}_{pushaccept} \cup \\ \mathcal{G}_{pushconflict} \cup \mathcal{G}_{popaccept} \cup \mathcal{G}_{popconflict} \cup \\ \texttt{Invoke}_\alpha \cup \texttt{Return}_\alpha \end{pmatrix}$$

$$\mathcal{R}[\alpha] \triangleq \bigcup_{\alpha' \neq \alpha} \mathcal{G}[\alpha']$$

Figure 24 Rely and guarantee definitions for the EB stack

$\{I \wedge \rho_C[\alpha] = \mathtt{push}(v)\}$
$\mathtt{push}(v)\{$
  $\{I \wedge \rho_C[\alpha] = \mathtt{push}(v)\}$
  $\mathtt{while}(\mathtt{true})\{$
    $\{I \wedge \rho_C[\alpha] = \mathtt{push}(v)\}$
    $b \leftarrow \mathtt{S.push}(v)$
    $\left\{\begin{matrix} I \wedge \rho_C[\alpha] = \mathtt{push}(v) \wedge \\ (b = \top \Rightarrow \rho_R[\alpha] = \mathtt{push}(v).\top) \end{matrix}\right\}$
    $\mathtt{if}(b = \top)\{$
      $\{I \wedge \rho_C[\alpha] = \mathtt{push}(v) \wedge \rho_R[\alpha] = \mathtt{push}(v).\top\}$
      $\mathtt{ret}\ \top$
    $\}\mathtt{else}\{$
      $\{I \wedge \rho_C[\alpha] = \mathtt{push}(v)\}$
      $\mathtt{E.exch}(v).\mathtt{inv}$
      $\left\{\begin{matrix} I \wedge \rho_C[\alpha] = \mathtt{push}(v) \wedge \\ \left(\begin{matrix} (\exists o, s_{\mathtt{exch}} = (o, \{\}) \wedge (\alpha, v) \in o) \vee \\ (\exists \beta w, s_{\mathtt{exch}} = (\{(\beta, w)\}, \{(\alpha, v)\})) \vee \\ \exists \beta, \left(\begin{matrix} s_{\mathtt{exch}} = (\{(\beta, \top)\}, \{(\alpha, v)\}) \wedge \\ \rho_R[\alpha] = \mathtt{push}(v).\top \end{matrix}\right) \end{matrix}\right) \end{matrix}\right\}$
      $r \leftarrow \mathtt{E.exch}(v).\mathtt{res}$
      $\left\{\begin{matrix} I \wedge \rho_C[\alpha] = \mathtt{push}(v) \wedge \\ (r = \top \Rightarrow \rho_R[\alpha] = \mathtt{push}(v).\top) \end{matrix}\right\}$
      $\mathtt{if}(r = \top)\{$
        $\{I \wedge \rho_C[\alpha] = \mathtt{push}(v) \wedge \rho_R[\alpha] = \mathtt{push}(v).\top\}$
        $\mathtt{ret}\ \top$
      $\}$
    $\}$
  $\}$
$\}$

Figure 25 EB stack proof: `push`

$\{I \wedge \rho_C[\alpha] = \texttt{pop}()\}$
$\texttt{pop}()\{$
  $\{I \wedge \rho_C[\alpha] = \texttt{pop}()\}$
  $\texttt{while}(\texttt{true})\{$
    $\{I \wedge \rho_C[\alpha] = \texttt{pop}()\}$
    $v \leftarrow \texttt{S.pop}()$
    $\{I \wedge \rho_C[\alpha] = \texttt{pop}() \wedge (v \neq \bot \Rightarrow \rho_R[\alpha] = \texttt{pop}().v)\}$
    $\texttt{if}(v \neq \bot)\{$
      $\{I \wedge \rho_C[\alpha] = \texttt{pop}() \wedge \rho_R[\alpha] = \texttt{pop}().v\}$
      $\texttt{ret}\, v$
    $\}\texttt{else}\{$
      $\{I \wedge \rho_C[\alpha] = \texttt{pop}()\}$
      $\texttt{E.exch}(\top)\texttt{.inv}$

$$\left\{\begin{array}{c} I \wedge \rho_C[\alpha] = \texttt{pop}() \wedge \\ \left(\begin{array}{c} (\exists o, s_{\texttt{exch}} = (o, \{\}) \wedge (\alpha, \top) \in o) \vee \\ (\exists \beta, s_{\texttt{exch}} = (\{(\beta, \top)\}, \{(\alpha, \top)\})) \vee \\ \exists \beta, \left(\begin{array}{c} s_{\texttt{exch}} = (\{(\beta, v)\}, \{(\alpha, \top)\}) \wedge \\ \rho_R[\alpha] = \texttt{pop}().v \end{array}\right) \end{array}\right) \end{array}\right\}$$

      $r \leftarrow \texttt{E.exch}(\top)\texttt{.res}$

$$\left\{\begin{array}{c} I \wedge \rho_C[\alpha] = \texttt{pop}() \\ \wedge (r \neq \top \wedge r \neq \bot \Rightarrow \rho_R[\alpha] = \texttt{pop}().r) \end{array}\right\}$$

      $\texttt{if}(r \neq \top \wedge r \neq \bot)\{$
        $\{I \wedge \rho_C[\alpha] = \texttt{pop}() \wedge \rho_R[\alpha] = \texttt{pop}().r\}$
        $\texttt{ret}\, r$
      $\}$
    $\}$
  $\}$
$\}$

Figure 26 EB stack proof: `pop`

B Detailed Comparison with Oliveira Vale et al. [26]

In this section we make a detailed comparison with [26], which our work is deeply rooted in. We hope this comparison will convince the reader that our methodology substantially extends their proposed verification technology, and that it provides further evidence of the efficacy of their techniques. We also note that the main contribution of this paper is the completeness proof for the program logic, which is completely novel even when compared with [26], and makes for the first complete program logic for any of the generalizations of linearizability beyond atomicity.

B.1 Trace-Based Reasoning vs. State-Based Reasoning

The major part of [26] deals with the theory of linearizability, which is typically presented in terms of traces. Because of this, likely in order to keep parity with the theory they describe, they present a program logic which is built around their game semantics, entirely based on traces.

Unfortunately, we found in our preliminary attempt to mechanize their work that their setup is not very practical. The issue is that sets of traces, while theoretically equivalent to state, result in a lot of extraneous reasoning. Most of the time, the extraneous reasoning comes from trying to establish some relationship between the traces and a state-transition system that simplifies the reasoning. The reader can check in [26] that their program logic proofs involve an excessive number of inductive predicates over traces to establish some correspondence with a notion of state, or some predicate that would be more easily formulated in terms of state.

Instead, we reworked their program logic to work with LTSs. As a result, the logic is more usable in our version, which results in less extraneous reasoning for the user. Many of us also believe it is more natural to work with state directly. As we discuss later in this section, this distinction leads to further improvements and simplifications in the reasoning in our program logic.

For instance, when verifying the ticket lock, they have to define the inductive predicate `newtkt` over traces of the underlay of type $\texttt{FAI} \otimes \texttt{Counter}$ to calculate the currently serving ticket. Because of this, proofs about these predicates, such as stability, will have to prove properties about the composition of several inductive predicates over traces, which is often quite convoluted. Meanwhile, for us the information calculated by `newtkt` is exactly the current state of the underlay `FAI` LTS, and hence involves no inductive reasoning.

B.2 Completeness

They do not prove the completeness of the program logic, which we remedy in our paper. While they claim that the axiomatic version of possibility reasoning is complete, they only *speculate* that the way they have encoded it in their program logic makes the program logic complete. *Prima facie*, there is no reason to believe this is the case, as there are certainly ways to make a rely-guarantee logic with similar reasoning principles that is not even complete for Herlihy-Wing linearizability, much less for the much more general notion of linearizability that [26] propose. [26] themselves present a negative result in that sense, showing that the program logic by [20] is not complete for Herlihy-Wing linearizability.

Our goal is to verify heterogeneous systems composed of objects with arbitrary degrees of concurrency. Completeness is a serious problem to address in this setting. For example, [20] leave as future work verifying the challenging Herlihy-Wing linearizable Timestamped

Stack [8]. In private communications with the first author of [20], they informed us that they believe it was actually either too challenging or impossible to verify the TS Stack in their program logic. If a researcher decided to pursue this line of future work, they would find themselves in a position where they would not know if they were struggling because of their own mistakes, or if the framework they are using is simply unable to verify the TS Stack. Now, with a program logic satisfying completeness, they would know that, because they were able to successfully encode their data structure implementation and its linearized specification in the framework, it is possible to prove it is correct in the framework.

Our program logic, while building upon the foundation laid out by [26], is proven *complete* for their definition of linearizability. This is a novel result that extends beyond a mere modification of their work. The completeness proof process led us to refine several program logic rules. One of the more pronounced changes is the use of a return commit, which we realized was necessary to prove completeness. Our codebase differs from their on-paper description in several ways, notably in our use of ITrees for program representation, which affects the underlying semantics, and the use of LTSs. Consequently, it remains unclear whether our refinements to the program logic stem from these fundamental modeling differences or represent necessary changes they would also need to implement to achieve completeness. Because they argue that their linearizability criterion is a complete safety property for verifying any concurrent object implemented by a module of parallel programs, our method is also a complete verification method for this rather large class of concurrent objects, which goes far beyond atomic and non-blocking objects (the class of objects characterized by Herlihy-Wing linearizability).

B.3 Possibility Representation

The way the representation of possibilities by [26] encodes a *set* of possible linearizations is very subtle. Their representation of possibilities is as a *single* concurrent trace s together with a side-condition (that the possibility trace is linearizable to the desired specification). A commit step in [26], beyond adding responses to pending invocations, consists of further linearizing the trace s according to a rewrite system. In other words, in a commit step, you may take the current possibility s and re-order it into a new possibility s' following their linearizability rewrite system. In early experiments in our mechanization, we have found that proofs of linearizability by re-ordering add a lot of LOC for the user in the process of using the program logic, while being obtuse to understand.

To illustrate some of these points, let us consider an example. Suppose you are verifying the Herlihy-Wing queue [15] and currently considering the concrete trace

$$s = \mathbf{1}{:}enq(1) \cdot \mathbf{2}{:}enq(2) \cdot \mathbf{1}{:}enq(1).ok \cdot \mathbf{2}{:}enq(2).ok$$

The possible linearizations of the trace s are (w.r.t. the usual linearized specification for a queue):

$$s_{12} = \mathbf{1}{:}enq(1) \cdot \mathbf{1}{:}enq(1).ok \cdot \mathbf{2}{:}enq(2) \cdot \mathbf{2}{:}enq(2).ok$$

$$s_{21} = \mathbf{2}{:}enq(2) \cdot \mathbf{2}{:}enq(2).ok \cdot \mathbf{1}{:}enq(1) \cdot \mathbf{1}{:}enq(1).ok$$

In a situation where the linearization of the enqueue is determined by the future, they would keep as possibility p exactly the trace $p = s$, as at the point the enqueues are running it might not be possible to determine which enqueue will be considered to have taken effect first. The reason for this is that in the Herlihy-Wing queue, there are executions where the

order of the enqueues is only determined when a dequeue runs. Suppose a dequeue happens and the dequeue will return 2 in the future. The current execution might be:

$$\mathbf{1}{:}\mathit{enq}(1) \cdot \mathbf{2}{:}\mathit{enq}(2) \cdot \mathbf{1}{:}\mathit{enq}(1).\mathit{ok} \cdot \mathbf{2}{:}\mathit{enq}(2).\mathit{ok} \cdot \mathbf{1}{:}\mathit{deq}()$$

Then they can complete the $\mathbf{1}{:}\mathit{deq}()$ call once they reach its linearization point, and evolve the possibility into

$$p' = \mathbf{2}{:}\mathit{enq}(2) \cdot \mathbf{2}{:}\mathit{enq}(2).\mathit{ok} \cdot \mathbf{1}{:}\mathit{enq}(1) \cdot \mathbf{1}{:}\mathit{enq}(1).\mathit{ok} \cdot \mathbf{1}{:}\mathit{deq}() \cdot \mathbf{1}{:}\mathit{deq}().2$$

Note that, in the process, they have linearized the events of p within the new possibility p'.

We argue that the way the single trace s corresponds to a set of possibilities leads to unnecessary complexity in the proofs. Without getting into the technical details, in their program logic the single trace p, as a possibility, implicitly encodes the set of all traces it linearizes to. In our experience, this makes the predicates necessary to describe the set of possibilities at each step too complicated. In some situations, *some* but *not all* of the linearizations of the possibility you'd have to keep are valid (see Appendix C.1 of [26] for an example). In such cases, the predicates in the program logic need to keep track of auxiliary information to differentiate which of the linearizations are still valid and which are not from the single trace p that is kept as the possibility. While technically correct, this results in convoluted reasoning in practice, and could make proving completeness more challenging for them.

On our side, we changed the framework to use LTSs for usability reasons. So, *a priori* it is already more natural for us to use a significantly different representation for the possibilities and an appropriately modified axiomatics for managing them. Note that there is no concept to replace the idea of "further linearizing the possibility" when the possibility is based on state (i.e., there is no notion of a state σ linearizing to a state σ'). In particular, the axiomatics for possibilities we described in our paper completely removes the need to reason about linearizability in terms of re-orderings (at least from the point of view of the user; we still have to re-establish the relationship between our possibility axiomatics and proofs of linearizability).

Our commit step, in contrast to theirs, consists of moving pending invocations/responses into the state component σ of our possibility triples $\langle \sigma, p_C, p_R \rangle$ by removing/adding them to the set of pending invocations (p_C)/responses (p_R) and taking a corresponding transition in the state component of the possibility.

In the example above, we would keep a set containing two possibilities: the possibility corresponding to s_{12} ($\langle [1,2], \varnothing, \varnothing \rangle$) and one corresponding to s_{21} ($\langle [2,1], \varnothing, \varnothing \rangle$), and those would be updated somewhat independently of each other on a commit step.

Beyond this, for related reasons, we also have to generalize their possibility reasoning principles to deal with a set of possibilities, instead of a single possibility (as it is necessary for completeness in our case). While they briefly mention it is possible to do so, they do not provide the generalization in their work. In the example above, once we see the dequeue take effect, we would drop the possibility ($\langle [1,2], [1 \mapsto \mathit{deq}], \varnothing \rangle$) from the set and would commit the dequeue to the other possibility, resulting in a set with only the following possibility: $\langle [1], [1 \mapsto \mathit{deq}], [1 \mapsto 2] \rangle$. Furthermore, in a case where only some but not all linearizations of their possibility p are valid, we are able to keep exactly the ones that are valid, as any non-empty set of possibilities may be used in our case.

We believe our way of representing and managing possibilities is more intuitive and practical than theirs, and results in simpler predicates and proofs. This comes at the expense of having a more complicated proof of soundness, as our possibility formalism (based on

possibility triples and state) is further from the original formalism used for linearizability (based on traces and rewriting). But for a mechanized program logic it is important to facilitate the reasoning for the user.

B.4 Key Differences in the Meta-Proofs

Most of their proofs for the compositionality of linearizability make use of the categorical structure they establish about their game model. These include locality, observational refinement, the property that implementations verified against linearizability specifications vertically compose, etc. This works quite homogeneously in their setup because both concurrent object specifications and implementations are represented by the same mathematical object: a concurrent strategy. This means that they live in the same (semi)category, and therefore enjoy the same composition operations.

The setup in our paper is heterogeneous in comparison, as is common in other verification frameworks. Modules (code) are represented by a different mathematical object (in this case, parameterized collections of ITrees) than concurrent object specifications (LTSs). There is a good reason for this: for the mathematical representation of code, we would like something in which a programming language can easily be interpreted, but for the specifications, we would like a representation that makes it easy to specify the traces an object generates while also being convenient to define predicates in the program logic. While this is not the only way to deal with this situation, it is one of the most common approaches in mechanized verification, as the choice of encoding is crucial for reducing the amount of work required in the mechanized setting.

Moreover, the usual categorical equations (such as the fact that the identity is a neutral element for composition) only hold up to equivalence in our model, as is expected of the kind of coinductive definition necessary to model while loops, for example. On the side of specifications, since the notion of behavior of an LTS is the set of traces it generates, many constructions only need to hold up to forward-backward simulation (think of representation independence results). As a result, as part of our work, we had to generalize their proofs to this more complex categorical setup, which is a novel aspect of our paper.

B.5 Mechanization and Examples

An obvious difference is that our work is entirely mechanized, while theirs is not. In particular, by mechanizing some of their theoretical results around compositionality and linearizability, we get further assurance of the correctness of these results.

In addition, while in principle [26]'s program logic could verify something like the EBStack or the Write-Snapshot object in a similar way to us, they have only verified the *mostly* atomic example involving a ticket lock and a coarse-grained locked queue (which we both verify and generalize to any atomic object). We add these interesting and more complex examples and do so in a mechanized fashion, which they had not done. The EBStack example is quite substantial, and makes quite a strong case for the compositionality that they advocate.

In particular, we believe we have the first mechanized proof of an interval-sequential object against its intended interval linearizability specification, as stated by the original paper [5]. We also argue that the exact way we compositionally verify the EBStack is novel, as we have argued in §1 and §6.