


1 A Complete Program Logic for Compositional 2 Linearizability

3 **Eashan Hatti** ✉ 


4 Yale University, New Haven, CT, USA

5 **Arthur Oliveira Vale** ✉ 


6 Yale University, New Haven, CT, USA

7 **Zhongye Wang** ✉ 

8 Yale University, New Haven, CT, USA

9 **Yueyang Feng** ✉ 

10 Yale University, New Haven, CT, USA

11 **Zhong Shao** ✉ 

12 Yale University, New Haven, CT, USA

13 — Abstract —

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

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

25 **2012 ACM Subject Classification** Theory of computation → Program verification; Computing
26 methodologies → Concurrent computing methodologies; Theory of computation → Programming
27 logic

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

30 **Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2026.3

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

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

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

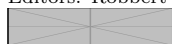
37 **Acknowledgements** We would like to thank the anonymous reviewers and artifact evaluation
38 committee for their thoughtful feedback and time spent reviewing our work.



© 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. 3:1–3:28



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

39 **1 Introduction**40 **1.1 Compositionality.**

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

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

55 In the context of concurrent objects, linearizability [15] has been the gold standard since
 56 the 90s. It allows programmers to abstract the behavior of a concurrent object E 's code M_E
 57 into a simpler specification V_E that is atomic¹ and easier to reason about. While there are
 58 many efforts to enable the mechanized verification of linearizability, few are truly scalable:
 59 either there are concurrent objects like the exchanger with non-atomic specifications that
 60 cannot be expressed in these pre-existing frameworks due to their over-reliance on atomicity,
 61 or the specification cannot be used as a black box encapsulating its code to modularly verify
 62 another object that uses this underlay object.

63 **1.2 Compositional Linearizability.**

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

73 While an important first step towards the compositional verification of non-atomic
 74 concurrent objects, and a convincing showcase of the compositional linearizability theory,
 75 the verification technique proposed by [26] could still be improved for the sake of practicality
 76 in large systems verification. Some immediate issues are that their framework has not been
 77 validated by a mechanization, and while [26] give a paper proof of soundness, they only
 78 conjecture its completeness. In this paper, we set out to mechanize the verification technique
 79 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.

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

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

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

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

102 Summary and Main Contributions

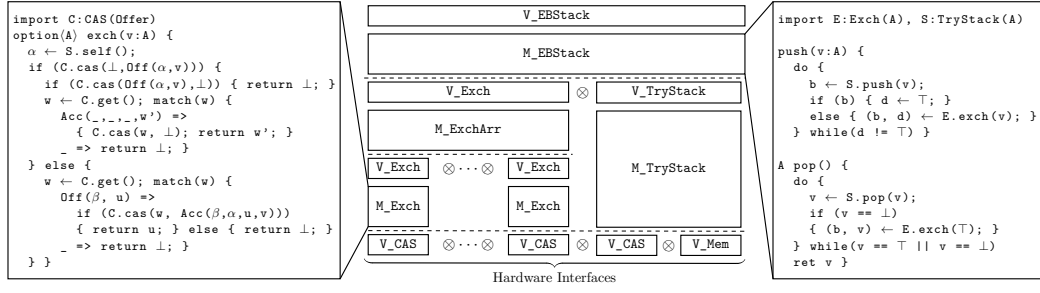
- 103 ■ We present the first *mechanized* program logic, Linearizability Hoare Logic (LHL), which
104 supports and unifies the verification of Herlihy-Wing linearizability and other non-atomic
105 linearizabilities into a generalized criterion called *compositional linearizability* [26].
- 106 ■ To mechanize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

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

122 **2** Motivating Example

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

3:4 A Complete Program Logic for Compositional Linearizability



■ **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 \perp value indicates the failure of the operation in both the exchanger and the `EBStack` object.

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

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

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

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

170 **3 Programming Language and Specifications**

171 Our framework verifies concurrent *modules* $M : \text{Mod } E \ F$, which implement operations
 172 specified by an *effect signature* F . A module M implements each operation $f \in F$ as a
 173 program $M^f : \text{Prog } E \ \text{ar}(f)$, which uses operations from an underlay signature E and
 174 returns a value in the return type (specified by $\text{ar}(f)$) of the operation f that it implements.
 175 An object over an effect signature E is specified by a state-transition system $V_E : \text{Spec } E$.
 176 The core verification task is to show that given a specification $V_E : \text{Spec } E$, called the
 177 *underlay* of M , the module $M : \text{Mod } E \ F$ correctly implements its overlay specification
 178 $V_F : \text{Spec } F$. We now discuss these notions formally.

179 **3.1 Effect Signatures**

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

$$184 \quad E := \{e : \text{par}_E(e) \rightarrow \text{ar}_E(e) \mid e \in E\}$$

185 When it causes no confusion, we omit the subscript on $\text{par}_E(-)$ and $\text{ar}_E(-)$.

In the context of our paper, effects $e \in E$ are method names, which take as arguments
 elements of the parameter set $\text{par}(e)$ and return some value in its arity $\text{ar}(e)$. 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.

$$EBStack := \{\text{push} : A \rightarrow \mathbf{1}, \text{pop} : \mathbf{1} \rightarrow A + \mathbf{1}\}$$

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

$$188 \quad E + F := \{\text{inl } e : \text{par}_E(e) \rightarrow \text{ar}_E(e) \mid e \in E\} \uplus \{\text{inr } e : \text{par}_F(e) \rightarrow \text{ar}_F(e) \mid e \in F\}$$

189 **3.2 Programs**

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

$$193 \quad p \in \text{Prog } E \ R := \text{Vis } f(a) \ k \mid \text{Ret } r \mid \text{Tau } p \text{ where } f \in E \quad a \in \text{par}(f) \quad k : \text{ar}(f) \rightarrow \text{Prog } E \ R \quad r \in R$$

3:6 A Complete Program Logic for Compositional Linearizability

194 A program $p \in \text{Prog } E R$ 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 $f(\mathbf{a}) \mathbf{k}$:** 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 $v \in \text{ar}(f)$.

198 **Ret r :** Once a program executes all of its operations, it terminates by returning a value r of
 199 type R .

200 **Tau \mathbf{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 $\text{Prog } E _$ 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 $\text{Prog } E _$. 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 $M : \text{Mod } E F$ implements operations of a signature F , using the operations of
 208 the signature E . A module $M : \text{Mod } E F$ consists of a mapping from operations $f \in F$ and
 209 arguments $a \in \text{par}(f)$ to programs $M^{f(a)} : \text{Prog } E \text{ar}(f)$, implementing a call $f(a)$ as a
 210 program operating over E and returning a value in its return type $\text{ar}(f)$. In other words,
 211 a module $M : \text{Mod } E F$ is a collection $M : \prod_{f \in F} (\text{par}(f) \rightarrow \text{Prog } E \text{ar}(f))$. For instance,
 212 M_{EBStack} is the mapping taking `push(v)` to the program described in Fig. 1 and similarly
 213 for `pop()`. Nonetheless, we write $M^{f(a)} : \text{Prog } E \text{ar}(f)$ for the program corresponding to
 214 $f \in F$ and $a \in \text{par}(f)$ in M , and $M^f : \text{par}(f) \rightarrow \text{Prog } E \text{ar}(f)$.

215 Modules may be composed vertically by interaction tree substitution. Given modules
 216 $M : \text{Mod } E F$ and $N : \text{Mod } F G$, their vertical composition $M : \triangleright N : \text{Mod } E G$ 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:

$$221 \quad \text{id}_M : \text{Mod } E E := (\text{id}_{\text{Prog}_e})_{e \in E} \quad \text{where } \text{id}_{\text{Prog}_e} := \lambda(a : \text{par}(e)). \text{Vis } e(a) \text{ Ret}$$

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

$$224 \quad (1) \text{id}_M : \triangleright M \approx M \quad (2) M : \triangleright \text{id}_M \approx M \quad (3) (M_1 : \triangleright M_2) : \triangleright M_3 \approx M_1 : \triangleright (M_2 : \triangleright M_3)$$

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 id_M 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 $M : \text{Mod } E F$ 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

236 $\{i \in \mathbb{N} \mid i < T\}$, where $T \in \mathbb{N}$ is the total number of threads, as \mathcal{T} . In a multicore setting,
 237 \mathcal{T} is the set of cores available; in a multi-threaded setting, it is the set of available threads.
 238 While users can fix the whole set \mathcal{T} when verifying an object, we choose to parameterize our
 239 theories and proofs over \mathcal{T} . As a result, our proofs of linearizability universally quantify over
 240 such finite sets of thread names.

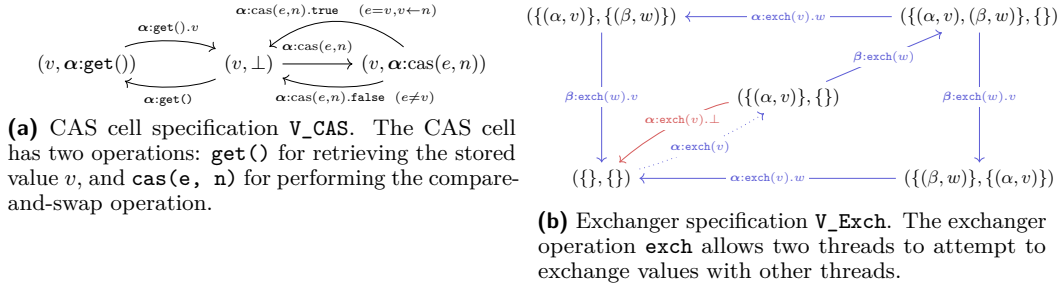
241 Then, an event of the signature \mathbf{E} consists of

242 $\mathbf{TEvent} \mathbf{E} := \alpha:f(a) \mid \alpha:f(a).v$

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

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

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



■ **Figure 2** CAS and Exchanger Linearized Specifications

260 ► **Example 1** (Atomic CAS Cell LTS). We begin with an atomic specification encoded by the
 261 LTS in Fig. 2a for a CAS cell object. Its states are tuples consisting of the stored value and
 262 the ongoing pending call. A non-empty pending call component restricts the next transition
 263 to be labelled with the matching return event. The two transitions on the left ensure that
 264 a get operation $\alpha:\text{get}()$ always gets the stored value v in the return event $\alpha:\text{get}().v$. The
 265 three transitions on the right specify that a CAS operation $\alpha:\text{cas}(e, n)$ receives **true** as a
 266 response only when the stored value is e and stores the new value n to it, and otherwise, it
 267 receives **false** and does not modify the stored value. This LTS specifies an atomic object
 268 because, after an invocation by thread α , the next transition must be a response by the same
 269 thread α . Invocation transitions are possible only when the ongoing operation is \perp , resulting
 270 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).

$$279 \quad \alpha:\text{exch}(v) \cdot \beta:\text{exch}(w) \cdot \alpha:\text{exch}(v).w \cdot \beta:\text{exch}(w).v \quad (1)$$

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

$$282 \quad \alpha:\text{exch}(v) \cdot \alpha:\text{exch}(v).\perp \quad (2)$$

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

$$\frac{s \in S}{s \xrightarrow{\epsilon}_{\mathbf{V}} s} \quad \frac{s_1 \xrightarrow{\text{ev}_{\mathbf{V}}} s_2}{s_1 \xrightarrow{\text{ev}} s_2} \quad \frac{s_1 \xrightarrow{t}_{\mathbf{V}} s_2 \quad s_2 \xrightarrow{t'}_{\mathbf{V}} s_3}{s_1 \xrightarrow{t \cdot t'}_{\mathbf{V}} s_3} \quad \llbracket \mathbf{V} \rrbracket := \{t \in (\mathbf{TEvent} \ \mathbf{E})^* \mid \exists s \in S. s_0 \xrightarrow{t}_{\mathbf{V}} s\}$$

Our notion of refinement is then just behavior containment:

$$\mathbf{V} \sqsupseteq \mathbf{V}' \iff \llbracket \mathbf{V} \rrbracket \subseteq \llbracket \mathbf{V}' \rrbracket \quad \mathbf{V} \equiv \mathbf{V}' \iff \mathbf{V} \sqsupseteq \mathbf{V}' \wedge \mathbf{V}' \sqsupseteq \mathbf{V}$$

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 $\mathbf{V_E} : \mathbf{Spec} \ \mathbf{E}$, where $\mathbf{V_E} = (S, \rightarrow_{\mathbf{V}}, s_0)$, and a module
 293 $M : \mathbf{Mod} \ \mathbf{E} \ \mathbf{F}$, we define the behaviors of M running on top of $\mathbf{V_E}$ operationally as a concrete
 294 LTS ($\mathbf{V_E} \triangleright M$). 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 \mathbf{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 $(ths, s) \in \mathbf{InterState}$ (interaction state) of the operational semantics are
 pairs of a *thread state* $ths \in \mathcal{T} \rightarrow \mathbf{TState} \ \mathbf{E} \ \mathbf{F}$, which is a map from thread names to the

$$\begin{array}{c}
\text{OCALL} \\
\frac{ths(\alpha) = \text{Idle}}{(ths, s) \xrightarrow{\alpha:f(a)} (ths[\alpha \mapsto \text{Cont } f(a) M^{f(a)}], s)} \\
\\
\text{UCALL} \\
\frac{ths(\alpha) = \text{Cont } f(a) (\text{Vis } g(b) k) \quad s \xrightarrow{\alpha:g(b)} s'}{(ths, s) \xrightarrow{\alpha:g(b)} (ths[\alpha \mapsto \text{UCall } f(a) g(b) k], s')} \\
\\
\text{URET} \\
\frac{ths(\alpha) = \text{UCall } f(a) g(b) k \quad s \xrightarrow{\alpha:g(b).v} s'}{(ths, s) \xrightarrow{\alpha:g(b).v} (ths[\alpha \mapsto \text{Cont } f(a) k(v)], s')} \\
\\
\text{SILENT} \\
\frac{ths(\alpha) = \text{Cont } f(a) (\text{Tau } p)}{(ths, s) \xrightarrow{\epsilon} (ths[\alpha \mapsto \text{Cont } f(a) p], s)}
\end{array}$$

■ **Figure 3** Operational Semantics

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

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

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

302 Fig. 3 defines the operational semantics rules. OCALL selects an idle thread, non-
303 deterministically chooses an operation $f \in F$ and an argument $a \in \text{par}(f)$, and changes the
304 continuation for α to the code for $f(a)$ defined as $M^{f(a)}$. UCALL locates a thread α whose
305 next step in the continuation is an underlay call $g(b)$, performs the call step $\alpha:g(b)$ in
306 the underlay object V_E , and updates the underlay LTS state to s' accordingly. URET is
307 analogous, except that it takes a return transition and continues with the continuation as
308 specified by k . ORET matches on a thread whose continuation indicates it is ready to return
309 from the overlay operation, and performs the overlay return by updating the continuation to
310 **Idle**. SILENT simply skips a **Tau** with no change to the state.

311 We then define the concrete LTS of running M on top of V_E as $V_E \triangleright M : \text{Spec } F :=$
312 $(\text{InterState } F \text{ } V_E, \rightarrow_{V_E \triangleright M}, (ths_0, s_0))$, where ths_0 is the constant mapping to **Idle** and
313 $m \in \text{TEvent } F$, and transitions are given by the following relation

$$\begin{array}{c}
314 (ths, s) \xrightarrow{m}_{V_E \triangleright M} (ths', s') \iff \exists ths'' \in \mathcal{T} \rightarrow \text{TState } E \text{ F}, s'' \in S, t \in (\text{TEvent } E)^* \\
315 (ths, s) \xrightarrow{m} (ths'', s'') \wedge (ths'', s'') \xrightarrow{t} (ths', s')
\end{array}$$

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

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

$$\begin{array}{c}
(1) M \approx M' \Rightarrow V \triangleright M \equiv V \triangleright M' \qquad (2) V' \sqsubseteq V \Rightarrow V' \triangleright M \sqsubseteq V \triangleright M \\
(3) V \triangleright (M : \triangleright N) \equiv (V \triangleright M) \triangleright N
\end{array}$$

3.7 Horizontal Composition

320 Both modules and specifications can be horizontally composed. For instance, given modules
321 $M_L : \text{Mod } E_L \text{ } F_L$ and $M_R : \text{Mod } E_R \text{ } F_R$ we can define their horizontal composition $M_L + M_R :$
322

3:10 A Complete Program Logic for Compositional Linearizability

323 $\text{Mod} (E_L + E_R) (F_L + F_R)$, similarly to how we did with effect signatures, as the union of the
 324 two collections:

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

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

For specifications $V_L : \text{Spec } E_L$ and $V_R : \text{Spec } E_R$ we can define a specification $V_L \otimes V_R$ 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 $- \otimes -$:

$$\frac{V_L \sqsupseteq V_L' \quad V_R \sqsupseteq V_R'}{V_L \otimes V_R \sqsupseteq V_L' \otimes V_R'} \quad \frac{V_L \otimes V_R \sqsupseteq V_L' \otimes V_R'}{V_L \sqsupseteq V_L'} \quad \frac{V_L \otimes V_R \sqsupseteq V_L' \otimes V_R'}{V_R \sqsupseteq V_R'}$$

$$\text{id}_{M_E} + \text{id}_{M_F} = \text{id}_{M_{E+F}}$$

$$(V_L \triangleright M_L) \otimes (V_R \triangleright M_R) \equiv (V_L \otimes V_R) \triangleright (M_L + M_R)$$

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 $\text{get} : \mathbf{1} \rightarrow \mathbb{N}$, taking unit as argument and returning some natural
 337 number, and $\text{inc} : \mathbf{1} \rightarrow \mathbf{1}$, 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:

$$340 \quad s = \text{inc} \cdot \text{ok} \cdot \text{get}$$

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:

$$343 \quad \alpha_1 : \text{inc} \cdot \alpha_1 : \text{ok} \cdot \alpha_1 : \text{get}$$

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

$$345 \quad \alpha_1 : \text{inc} \cdot \alpha_1 : \text{ok} \cdot \alpha_1 : \text{get} \cdot \alpha_2 : \text{inc} \cdot \alpha_2 : \text{ok} \dots \alpha_k : \text{inc} \cdot \alpha_k : \text{ok} \cdot \alpha_1 : n$$

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.

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

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

$$373 \quad I_1 \cdot R_1 \cdot I_2 \cdot R_2 \cdot \dots \cdot I_n \cdot R_n$$

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

$$377 \quad I_1 \cdot R_1 \cdot I_2 \cdot R_2 \cdot \dots \cdot I_n \cdot R_n$$

378 is in the specification, then so is

$$379 \quad I'_1 \cdot R'_1 \cdot I'_2 \cdot R'_2 \cdot \dots \cdot I'_n \cdot R'_n$$

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

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

$$390 \quad s' = \alpha:\text{ws}(3) \cdot \beta:\text{ws}(5) \cdot \alpha:\text{ws}(3).\{3, 5\} \cdot \gamma:\text{ws}(2) \cdot \beta:\text{ws}(5).\{3, 5, 2\}$$

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

3:12 A Complete Program Logic for Compositional Linearizability

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

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

$$397 \quad I_1 \cdot R_1 \cdot I_2 \cdot R_2 \cdot \dots \cdot I_n \cdot R_n$$

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

$$406 \quad I_1 \cdot R_1 \cdot \dots \cdot I_k$$

407 then in any two extensions

$$408 \quad I_1 \cdot R_1 \cdot \dots \cdot I_k \cdot R_k \quad \text{and} \quad I_1 \cdot R_1 \cdot \dots \cdot I_k \cdot R'_k$$

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

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

We say a concrete LTS $V' : \text{Spec } E$ is linearizable w.r.t. $V : \text{Spec } E$, 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 $V \triangleright \text{idM}$, 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 $V_{_E'}, V_{_E} \in \text{Spec } E$, $V_{_F'}, V_{_F} \in \text{Spec } F$, and $M \in \text{Mod } E \ G$, we have observational refinement (1) and locality (2):*

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

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

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 $(V_{_E1} \triangleright M_{_F1}) \rightsquigarrow V_{_F1}$ and $(V_{_E2} \triangleright M_{_F2}) \rightsquigarrow V_{_F2}$, then $((V_{_E1} \otimes V_{_E2}) \triangleright (M_{_F1} \otimes M_{_F2})) \rightsquigarrow (V_{_F1} \otimes V_{_F2})$.*

► **Theorem 6** (Vertical Comp.). *If $(V_{_E} \triangleright M_{_E}) \rightsquigarrow V_{_F}$ and $(V_{_F} \triangleright M_{_F}) \rightsquigarrow V_{_G}$, then $(V_{_E} \triangleright (M_{_E} \triangleright M_{_F})) \rightsquigarrow V_{_G}$.*

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 $F_1 + F_2$ 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 $\rho \in \text{Poss}$ 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; $\rho_C(\alpha) = \perp$ 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 $\rho_C(\alpha) = \perp$ and $\rho_R(\alpha) = \perp$.

507 **Pending Call:** α ’s last transition in V_F was a return, $\rho_C(\alpha) = m$ for some $m \in F$ and
 508 $\rho_R(\alpha) = \perp$.

509 **Call Done:** The last transition by α in V_F was a call $\alpha:m$, $\rho_C(\alpha) = m$ and $\rho_R(\alpha) = \perp$.

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

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:

$$\begin{array}{c}
 \text{INVOKE} \\
 \frac{\rho_C(\alpha) = \perp \quad \rho_R(\alpha) = \perp}{\langle \rho_S, \rho_C, \rho_R \rangle \dashrightarrow \langle \rho_S, \rho_C[\alpha \mapsto m], \rho_R \rangle} \\
 \\
 \text{CCOMMIT} \\
 \frac{\rho_S \xrightarrow{\alpha:m} t \quad \rho_C(\alpha) = m \quad \rho_R(\alpha) = \perp}{\langle \rho_S, \rho_C, \rho_R \rangle \dashrightarrow \langle t, \rho_C, \rho_R \rangle} \\
 \\
 \text{RETURN} \\
 \frac{\rho_C(\alpha) = m \quad \rho_R(\alpha) = n}{\langle \rho_S, \rho_C, \rho_R \rangle \dashrightarrow \langle \rho_S, \rho_C[\alpha \mapsto \perp], \rho_R[\alpha \mapsto \perp] \rangle} \\
 \\
 \text{RCOMMIT} \\
 \frac{\rho_S \xrightarrow{\alpha:m.n} t \quad \rho_C(\alpha) = m \quad \rho_R(\alpha) = \perp}{\langle \rho_S, \rho_C, \rho_R \rangle \dashrightarrow \langle t, \rho_C, \rho_R[\alpha \mapsto n] \rangle} \\
 \\
 \bar{\rho} \rightarrow \bar{\sigma} \iff \forall \sigma \in \bar{\sigma}. \exists \rho \in \bar{\rho}. \rho \dashrightarrow^* \sigma
 \end{array}$$

■ **Figure 4** Possibility update rules

	$\langle ([], \perp), \emptyset, \emptyset \rangle$
\rightarrow (INVOKE)	$\langle ([], \perp), [\alpha_1 \mapsto \text{push}(1)], \emptyset \rangle$
\rightarrow (INVOKE)	$\langle ([], \perp), [\alpha_1 \mapsto \text{push}(1)][\alpha_2 \mapsto \text{push}(2)], \emptyset \rangle$
\rightarrow (CCOMMIT)	$\langle ([], \alpha_i:\text{push}(i)), [\alpha_1 \mapsto \text{push}(1)][\alpha_2 \mapsto \text{push}(2)], \emptyset \rangle$
\rightarrow (RCOMMIT)	$\left\langle (i :: [], \perp), \left[\begin{array}{l} \alpha_1 \mapsto \text{push}(1), \\ \alpha_2 \mapsto \text{push}(2) \end{array} \right], [\alpha_i \mapsto \text{push}(i).()] \right\rangle$
\rightarrow (CCOMMIT)	$\left\langle \left(\begin{array}{l} i :: [], \\ \alpha_j:\text{push}(j) \end{array} \right), \left[\begin{array}{l} \alpha_1 \mapsto \text{push}(1), \\ \alpha_2 \mapsto \text{push}(2) \end{array} \right], [\alpha_i \mapsto \text{push}(i).()] \right\rangle$
\rightarrow (RCOMMIT)	$\left\langle (j :: i :: [], \perp), \left[\begin{array}{l} \alpha_1 \mapsto \text{push}(1), \\ \alpha_2 \mapsto \text{push}(2) \end{array} \right], \left[\begin{array}{l} \alpha_i \mapsto \text{push}(i).(), \\ \alpha_j \mapsto \text{push}(j).() \end{array} \right] \right\rangle$
\rightarrow (RETURN)	$\langle (j :: i :: [], \perp), [\alpha_j \mapsto \text{push}(j)], [\alpha_j \mapsto \text{push}(j).()] \rangle$
\rightarrow (RETURN)	$\langle (j :: i :: [], \perp), \emptyset, \emptyset \rangle$
\rightarrow (INVOKE)	$\langle (j :: i :: [], \perp), [\alpha_3 \mapsto \text{pop}()], \emptyset \rangle$
\rightarrow (CCOMMIT)	$\langle (j :: i :: [], \alpha_3:\text{pop}()), [\alpha_3 \mapsto \text{pop}()], \emptyset \rangle$
\rightarrow (RCOMMIT)	$\langle (i :: [], \perp), [\alpha_3 \mapsto \text{pop}()], [\alpha_3 \mapsto \text{pop}().j] \rangle$

■ **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.

$$t' = \alpha_1:\text{push}(1) \cdot \alpha_2:\text{push}(2) \cdot \alpha_1:\text{push}(1).() \cdot \alpha_2:\text{push}(2).() \cdot \alpha_3:\text{pop}()$$

$$t_{i,j} = \alpha_i:\text{push}(i) \cdot \alpha_i:\text{push}(i).() \cdot \alpha_j:\text{push}(j) \cdot \alpha_j:\text{push}(j).() \cdot \alpha_3:\text{pop}() \cdot \alpha_3:\text{pop}().j$$

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 $\bar{\rho}$, and define the lifted transition \rightarrow to \rightarrow .³ In other words, a set of possibilities $\bar{\rho}$ 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 $\bar{\sigma}$ as long as every possibility in $\bar{\sigma}$ is reachable after zero or finitely many valid updates
 537 from some possibility in $\bar{\rho}$.

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 $\bar{\rho} = \{\rho_{1,2}, \rho_{2,1}\}$ 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 $\bar{\rho}$ 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:

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

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 `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 $(s, \bar{\rho})$ of a concrete LTS state $s \in \text{InterState}$ and
 563 a set of possibilities $\bar{\rho} \in \mathcal{P}(\text{Poss})$. The component s represents the current state of the
 564 program, which follows the actual semantics of the program, while $\bar{\rho}$ 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 $V_E, V_F, \mathcal{R}, \mathcal{G}, P, Q \models M$ groups
577 all side conditions and implies our linearizability condition $(V_E \triangleright M_F) \rightsquigarrow V_F$, (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 $f \in F$
and rely/guarantee relations, and show the following:

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

582 where $\text{Inv}(-) := \cup_{f \in F} \text{Inv}(f)$, $\text{Ret}(-) := \cup_{f \in F} \text{Ret}(f)$. 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.

$$\begin{array}{l} 598 \quad I(s, \rho) := \exists \alpha, \beta, v, w. \text{offered}(\alpha, v)(s, \rho) \vee \text{accepted}(\alpha, \beta, v, w)(s, \rho) \vee \text{cleared}(s, \rho) \\ 599 \quad \text{offered}(\alpha, v)(s, \rho) := s_{\text{CAS}} = \text{Off}(\alpha, v) \wedge \rho_S = (\{\}, \{\}) \wedge \rho_C(\alpha) = \text{exch}(v) \wedge \rho_R(\alpha) = \perp \\ 600 \quad \text{accepted}(\alpha, \beta, v, w)(s, \rho) := s_{\text{CAS}} = \text{Acc}(\alpha, \beta, v, w) \wedge \rho_S = (\{\}, \{\}) \\ 601 \quad \quad \quad \wedge \rho_C(\alpha) = \text{exch}(v) \wedge \rho_R(\alpha) = \text{exch}(v).w \\ 602 \quad \text{cleared}(s, \rho) := s_{\text{CAS}} = \perp \wedge \rho_S = (\{\}, \{\}) \end{array}$$

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:

$$\frac{P \implies Q \ v}{\mathcal{R}, \mathcal{G} \vdash \{P\} \text{Ret } v \{Q\}} \text{RET}$$

$$\frac{\text{Stable}(\mathcal{R}, Q_S) \quad \mathcal{R}, \mathcal{G} \vdash \{P; Q_S\} p \{Q\} \quad P \xrightarrow{\alpha; \epsilon} \lambda s, t, \bar{\rho}. Q_S(s, \bar{\rho}, t, \bar{\rho}) \wedge \mathcal{G}(s, \bar{\rho}, t, \bar{\rho})}{\mathcal{R}, \mathcal{G} \vdash \{P\} \text{Tau } p \{Q\}} \text{SILENT}$$

$$\frac{\mathcal{G} \vdash_{\alpha} \{P\} g(x) \{Q_C\} \quad \forall v. \mathcal{G} \vdash_{\alpha} \{P; Q_C\} g(x).v \{Q_R(v)\} \quad \text{Stable}(\mathcal{R}, Q_C) \quad \forall v. \text{Stable}(\mathcal{R}, Q_R(v)) \quad \forall v. \mathcal{R}, \mathcal{G} \vdash \{P; Q_C; Q_R(v)\} k(v) \{Q\}}{\mathcal{R}, \mathcal{G} \vdash \{P\} \text{Vis } g(x) k \{Q\}} \text{UNDERLAYSTEP}$$

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

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.

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

$$\frac{\text{Stable}(\mathcal{R}, Q) \quad \text{Stable}(\mathcal{R}, R) \quad \mathcal{G} \vdash_{\alpha} \{P\} g(x) \{Q\} \quad \forall v. \mathcal{G} \vdash_{\alpha} \{P; Q\} g(x).v \{R(v)\}}{\mathcal{R}, \mathcal{G} \vdash \{P\} g(x) \{P; Q; R\}}$$

 606 **5.5 Commit Judgment**

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

$$\frac{P \xrightarrow{\alpha; m} \lambda s, t, \bar{\rho}. \exists \bar{\sigma}. \bar{\rho} \twoheadrightarrow^* \bar{\sigma} \wedge Q(s, \bar{\rho}, t, \bar{\sigma}) \wedge \mathcal{G}(s, \bar{\rho}, t, \bar{\sigma})}{\mathcal{G} \vdash_{\alpha} \{P\} m \{Q\}} \text{COMMIT}$$

610 The commit judgment asks the following: for any state $(s, \bar{\rho})$ satisfying the pre-condition
 611 P and any t reachable from s through the operational semantics, the user needs to produce
 612 new linearizations $\bar{\sigma}$ reachable after finitely many possibility updates ($\bar{\rho} \twoheadrightarrow^* \bar{\sigma}$, 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 $\bar{\rho}$ 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:

$$\begin{aligned} & \{\text{offered}(\alpha, v)(s, \rho) \vee \exists \beta w. \text{accepted}(\alpha, \beta, v, w)(s, \rho)\} \\ & \text{if}(C.\text{cas}(\text{Off}(\alpha, v), \perp)) \{ \\ & \quad \left\{ \begin{array}{l} \text{cleared}(t, \sigma) \vee (\exists \beta w. \text{offered}(\beta, w)(t, \sigma)) \\ \vee \exists \gamma u. \text{accepted}(\beta, \gamma, w, u)(t, \sigma) \end{array} \right\} \wedge \rho_R(\alpha) = \text{exch}(v). \perp \} \end{aligned}$$

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

$$628 \quad \forall s, \rho, t. \left(\begin{array}{l} \text{offered}(\alpha, v)(s, \rho) \vee \exists \beta w. \text{accepted}(\alpha, \beta, v, w)(s, \rho) \\ \wedge s \xrightarrow{\alpha: \text{cas}(\text{Off}(\alpha, v)). \text{true}, \perp} t \end{array} \right)$$

$$629 \quad \Rightarrow \exists \sigma. \left(\begin{array}{l} \rho \xrightarrow{\alpha: \text{exch}(v), \alpha: \text{exch}(v). \perp} \sigma \wedge \mathcal{G}(s, \rho, t, \sigma) \wedge \\ \text{cleared}(t, \sigma) \vee (\exists \beta w. \text{offered}(\beta, w)(t, \sigma)) \\ \vee \exists \gamma u. \text{accepted}(\beta, \gamma, w, u)(t, \sigma) \end{array} \right)$$

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

637 Formally, the following rely-guarantee definition justifies this update, which matches the
638 $\mathcal{G}_{\text{fail}}$ relation.

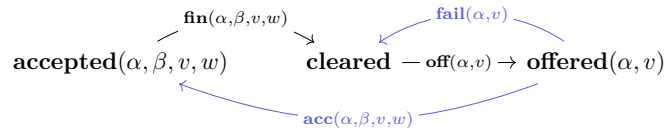
$$639 \quad \mathcal{G}_{\text{fail}}(\alpha, v)(s, \rho, t, \sigma) := \left(\begin{array}{l} s \xrightarrow{\alpha: \text{cas}(a, \perp). \text{true}} t \\ \wedge \rho \xrightarrow{\alpha: \text{exch}(v), \alpha: \text{exch}(v). \perp} \sigma \end{array} \right)$$

$$640 \quad \mathcal{G}_{\text{acc}}(\alpha, \beta, v, w)(s, \rho, t, \sigma) := \left(\begin{array}{l} s \xrightarrow{\beta: \text{cas}(\text{Off}(\alpha, v), \text{Acc}(\alpha, \beta, v, w)). \text{true}} t \\ \wedge \rho \xrightarrow{\alpha: \text{exch}(v), \beta: \text{exch}(w), \alpha: \text{exch}(v). w, \beta: \text{exch}(w). v} \sigma \end{array} \right)$$

$$641 \quad \mathcal{G}[\alpha] \triangleq \mathcal{G}_{\text{fail}}(\alpha, -) \cup \mathcal{G}_{\text{acc}}(\alpha, -, -, -) \cup \mathcal{G}_{\text{off}}(\alpha, -) \cup \mathcal{G}_{\text{fin}}(\alpha)$$

$$642 \quad \mathcal{R}[\alpha] \triangleq \cup_{\alpha' \neq \alpha} \mathcal{G}[\alpha']$$

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



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

644 accept transition \mathcal{G}_{acc} , so the precondition in the Hoare triple is stable. The post-condition
645 is also stable because no guarantee condition can step outside the invariant.
646

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

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

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

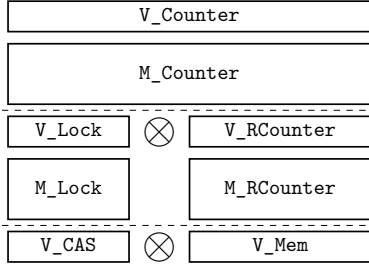
649 **5.6 Further Examples**

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

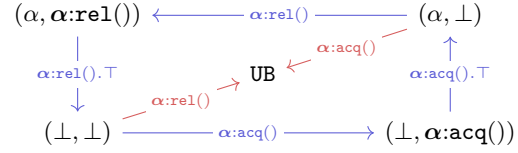
 654 **5.6.1 Lock-Protected Objects**

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

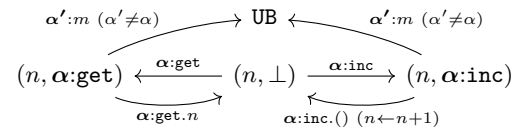
661 For example, Fig. 7 shows the structure of an atomic counter built above the lock and
 662 a racy counter. The racy counter specification (Fig. 9) adds an undefined behavior state
 663 (UB) to the atomic version, which would be entered whenever a concurrent access takes
 664 place. While in the UB state, any transition back into UB is allowed (as long as it does not
 665 break local sequentiality). The main proof obligation in verifying the atomic counter, i.e.,
 666 $((V_Lock \otimes V_RCounter) \triangleright M_Counter) \rightsquigarrow V_Counter$, is to ensure that the racy counter
 667 would never enter the UB state. Thanks to observational refinement, the lock specification
 668 (Fig. 8) ensures the atomic counter would never observe concurrent acqs – we gain mutual
 669 exclusion of $\text{acq} - \text{rel}$ pairs “for free”. As racy counter accesses only occur inside the critical
 670 region, concurrent counter accesses are never observed, and thus we show the overlay counter
 671 is safe and atomic.



672 **Figure 7** Module structure of the atomic counter. $RCounter$ and $Counter$
 673 denote the racy counter and the atomic counter, respectively.



674 **Figure 8** Lock specification



675 **Figure 9** Racy counter specification. Atomic counter
 676 specification can be obtained by removing the UB state.

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

674 **► Theorem 9.** *If V_E is an atomic LTS (invocations are immediately followed by responses
 675 of the same thread), $Racy(V_E)$ is its racy version (where concurrent invocations lead to
 676 the UB state), and M_E lifts operations in $Racy(V_E)$ with $\text{acq} - \text{rel}$ pairs, then $((V_Lock \otimes$
 677 $Racy(V_E)) \triangleright M_E) \rightsquigarrow V_E$.*

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

$$\alpha:\text{ws}(v) (\alpha \notin p, p \leftarrow p \cup \{\alpha\}, vs \leftarrow vs \cup \{v\}) \left((vs, p) \right) \beta:\text{ws}(v).vs (\beta \in p)$$

■ **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:

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

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 $V_E, V_F, \mathcal{R}, \mathcal{G}, P, Q \models M$, then $V_E \triangleright M \rightsquigarrow V_F$.*

Proof. Recall that $V_E \triangleright M$ 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 $V_E \triangleright M \rightsquigarrow V_F$, we can show that for every transition in $V_E \triangleright M$, 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 $V_E \triangleright M$ 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, \bar{p}) :

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

$$\begin{aligned} & \mathcal{R}[\alpha], \mathcal{G}[\alpha] \vdash \{P[\alpha]_f(s, \bar{p}); \text{Inv}(f); I\} p \{Q[\alpha]_f\} \\ & \wedge (P[\alpha]_f(s, \bar{p}); \text{Inv}(f); I)(s, \bar{p}) \wedge \text{Stable}(\mathcal{R}[\alpha]_f, I) \end{aligned}$$

3. If the thread state of α is `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.

$$\begin{aligned} & \mathcal{R}[\alpha], \mathcal{G}[\alpha] \vdash \{\mathcal{G}[\alpha]_f \vdash_\alpha \{P[\alpha]_f(s, \bar{p}); \text{Inv}(f); I\} g \{S\} \\ & \wedge P[\alpha]_f(s, \bar{p}); \text{Inv}(f); I; S\} p \{Q[\alpha]_f\} \wedge \text{Stable}(\mathcal{R}[\alpha]_f, I) \\ & \wedge \text{Stable}(\mathcal{R}[\alpha]_f, S) \wedge (P[\alpha]_f(s, \bar{p}); \text{Inv}(f); I)(s, \bar{p}) \end{aligned}$$

The proof then steps through the execution of $V_E \triangleright M$, 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 $P[\alpha]_f$; $\text{Inv}(f)$ and postcondition $Q[\alpha]_f$. $\text{Inv}(f)$ 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 $V_E \triangleright M \rightsquigarrow V_F$. ◀

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

$$\exists \mathcal{R} \mathcal{G} P Q. \forall_{E, F} \mathcal{R}, \mathcal{G}, P, Q \models M$$

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 $\alpha:m$. As a result,
 725 given a sequence of possibility updates q , we may take its projection $q \upharpoonright_{-,F}$ to the overlay
 726 events implied by the rule applications (INVOKE and RETURN rules), or its projection $q \upharpoonright_{E,-}$
 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 $I(s, \bar{p})$, 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 \bar{p} , ρ is reachable through possibility updates q with the same
 734 overlay events as p (i.e., $q \upharpoonright_{-,F} = p \upharpoonright_{-,F}$).
- 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., $q \upharpoonright_{-,F} = p \upharpoonright_{-,F}$), 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:

$$P(s, \bar{p}) \iff I(s, \bar{p}) \quad Q(_, _, t, \bar{\sigma}) \iff I(t, \bar{\sigma}) \quad \mathcal{R}(s, \bar{p}, t, \bar{\sigma}) \iff I(s, \bar{p}) \Rightarrow I(t, \bar{\sigma})$$

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

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 ($V \triangleright M$), 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 $\langle P \rangle_{\text{op}} \langle Q \rangle$ 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 $M_E \sqsubseteq C A_E$. 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

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

838 6.6 Relational Hoare logic

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

851 [30] encodes the contextual refinement obligation into a ghost state to verify it with
852 CSL. Iris [18] is a higher-order logic for verifying fine-grained concurrent programs and
853 also features logical atomicity. It leads to the ReLoC framework [10, 11, 32], which shows
854 contextual refinement of concurrent programs using a program logic, with prophecy variables
855 [19] enabling future-dependent linearization. ReLoC achieves compositionality by first
856 encapsulating reusable code into smaller functions, then verifying these functions against
857 logically atomic Hoare triples encoded as a logical relation, and lastly verifying the refinement
858 of the top-level object using these intermediate proofs. However, it is unknown how to reuse
859 the top-level refinement specification when verifying objects that use this top-level object.
860 As a result, every time they build new objects using the one verified against the refinement
861 specification, they need to prove a different functional correctness specification before they
862 can reuse its proof. In contrast, our framework imposes LTS specifications uniformly at any
863 level of abstraction.

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

875 7 Evaluation and Conclusion

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

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

		EBStack	Exchanger	Elimination Array	TryStack*
	R-LRG[22]	●	○	○	○
	ReLoC[11]	●	○	○	○
	LHL	●	●	●	●
LOC of LHL in Rocq	Spec	50	38	0 [†]	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.

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

895 — References —

- 896 1 Nick Benton. Simple relational correctness proofs for static analyses and program transforma-
 897 tions. *ACM SIGPLAN Notices*, 39(1):14–25, 2004. doi:10.1145/964001.964003.
- 898 2 Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen,
 899 and Nikos Tzevelekos. Theorems for free from separation logic specifications. *Proceedings of*
 900 *the ACM on Programming Languages*, 5(ICFP):1–29, 2021. doi:10.1145/3473586.
- 901 3 Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming (extended
 902 abstract). In Jim Anderson and Sam Toueg, editors, *Proceedings of the Twelfth Annual ACM*
 903 *Symposium on Principles of Distributed Computing, Ithaca, New York, USA, August 15-18,*
 904 *1993*, pages 41–51. ACM, 1993. doi:10.1145/164051.164056.
- 905 4 Jeremy S Bradbury, James R Cordy, and Juergen Dingel. Mutation operators for concurrent
 906 java (j2se 5.0). In *Second Workshop on Mutation Analysis (Mutation 2006-ISSRE Workshops*
 907 *2006)*, pages 11–11. IEEE, 2006.
- 908 5 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Specifying concurrent problems:
 909 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/1h1.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.](https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.612221)
997 [612221](https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.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.