C4: Verified Transactional Objects

MOHSEN LESANI, University of California, Riverside, USA
LI-YAO XIA, University of Pennsylvania, USA
ANDERS KASEORG, Massachusetts Institute of Technology, USA
CHRISTIAN J. BELL, Massachusetts Institute of Technology, USA
ADAM CHLIPALA, Massachusetts Institute of Technology, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA
STEVE ZDANCEWIC, University of Pennsylvania, USA

Transactional objects combine the performance of classical concurrent objects with the high-level programmability of transactional memory. However, verifying the correctness of transactional objects is tricky, requiring reasoning simultaneously about classical concurrent objects, which guarantee the atomicity of individual methods—the property known as linearizability—and about software-transactional-memory libraries, which guarantee the atomicity of user-defined sequences of method calls—or serializability.

We present a formal-verification framework called C4, built up from the familiar notion of linearizability and its compositional properties, that allows proof of both kinds of libraries, along with composition of theorems from both styles to prove correctness of applications or further libraries. We apply the framework in a significant case study, verifying a transactional set object built out of both classical and transactional components following the technique of *transactional predication*; the proof is modular, reasoning separately about the transactional and nontransactional parts of the implementation. Central to our approach is the use of syntactic transformers on *interaction trees*—i.e., transactional libraries that transform client code to enforce particular synchronization disciplines. Our framework and case studies are mechanized in Coq.

CCS Concepts: • **Software and its engineering** → **Software libraries and repositories**; • **Theory of computation** → **Program verification**.

Additional Key Words and Phrases: concurrency, objects, linearizability, serializability, verification

ACM Reference Format:
Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. 2022. C4: Verified Transactional Objects. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 80 (April 2022), 31 pages. https://doi.org/10.1145/3527324

1 INTRODUCTION

Two styles of concurrency have been studied intensively from a formal-methods perspective. On the one hand, we have classic data structures (e.g., stacks, queues, dictionaries) that rely on primitives like locks and compare-and-set instructions to guarantee atomicity of their methods; we might call these *single-method-atomic* data structures. On the other hand, there is a separate tradition of

Authors' addresses: Mohsen Lesani, University of California, Riverside, USA; Li-yao Xia, University of Pennsylvania, USA; Anders Kaseorg, Massachusetts Institute of Technology, USA; Christian J. Bell, Massachusetts Institute of Technology, USA; Adam Chlipala, Massachusetts Institute of Technology, USA; Benjamin C. Pierce, University of Pennsylvania, USA; Steve Zdancewic, University of Pennsylvania, USA.

2475-1421/2022/4-ART80
https://doi.org/10.1145/3527324

transaction-based APIs, which guarantee atomicity of chains of method calls; let us call them *multi-method-atomic*. In broad strokes, single-method atomicity is appealing for its superior performance, while multi-method atomicity is appealing for its simpler programming model.

Sometimes it is useful to break a complex concurrent application into pieces written in *both* of these styles [Assa et al. 2020; Elizarov et al. 2019; Spiegelman et al. 2016], but no one had previously shown how to prove functional correctness of such applications. We remedy this gap with a new framework for modular proofs of programs that mix the two styles. Proofs in this domain are intricate enough that machine checking is invaluable, so we present our framework as a Coq library.

As a motivating case study, we focus on *transactional predication* [Bronson et al. 2010]. The idea behind data structures implemented in this style—typically finite sets or maps—is to combine software transactional memory (STM) [Harris et al. 2005; Shavit and Touitou 1995], which provides good compositionality and reasoning properties but is relatively inefficient, with a concurrent data structure that exhibits good performance but provides a noncomposable interface. Instead of storing the entire data structure in transactional memory, we store *predicates* about the data structure—Boolean-valued mutable references indicating the membership of particular elements in the set. Updates to the predicates reflect actual changes to the data structure, while a nontransactional concurrent object manages the mapping between keys and predicates; this split reduces conflict detection to the STM's detection of write-write and read-write collisions. The challenge is finding a framework that supports ergonomic proofs involving both kinds of mechanisms.

A growing community of "programmer-provers" have learned to be effective at proving properties of code written in the native programming languages of various proof assistants. These native languages tend to be purely functional, but frameworks like *interaction trees* [Xia et al. 2020] have demonstrated how to extend the lightweight combination of programming and proof that arises naturally for pure functional programs to situations involving a range of computational effects. Unfortunately, when we include concurrency the complexity of reasoning increases dramatically.

To add concurrency on top of a proof assistant's native functional language and proof tools, we organize code into layered collections of concurrent objects whose specifications force all methods to behave atomically; in this setting we support a variety of concurrency styles, principally those associated with traditional linearizable concurrent data structures and serializable transactional memory. To support transactions, we introduce a novel technique of *passing code reified as interaction trees* and applying instrumentation functions to transform those trees.

For example, consider a concurrent stack object with methods *push* and *pop*. If such an object has been proven *linearizable* against a natural specification, clients can treat calls to *push* and *pop* as atomic, ignoring their actual implementations, which might use tricky fine-grained primitives like compare-and-swap. However, one level up, clients *do* still need to reason about the ways that sequences of calls to *push* and *pop* can nondeterministically interleave. Consider the following function, which transfers the elements of one concurrent stack to another:

```
moveStack(from, to) := vo ← from.pop();
                       match vo with None ⇒ return ()
                                   | Some(v) ⇒ moveStack(from, to); to.push(v)
```

Multiple calls to `moveStack` from different client threads may interleave their invocations of `push` and `pop`.

The complexity of reasoning about concurrent objects is one reason why *transactions* [Harris et al. 2005; Herlihy and Moss 1993; Papadimitriou 1979; Shavit and Touitou 1995] have become a popular concurrency abstraction. Transactions allow the programmer to declare that a particular block of code must be run atomically, leaving it to the compiler and/or runtime protocol to figure out how to provide this atomicity both soundly and efficiently.

Software transactional memory is probably the best-known realization of this idea in the functional-programming world. It allows programmers to write code like

$$moveStackAtomically(from, to) \;:=\; asTransaction(\lambda\_.\ moveStack(from, to))$$

where the call to *moveStack* is wrapped in a thunk and passed to a library procedure that promises to run such thunks atomically.

Our goal is to blend transactions and classical concurrent objects into a unified concurrent-object framework. In particular, we want to capture transaction protocols as objects that accept user transactions—thunks that chain calls together—and execute them atomically. Further, we want to express the correctness of such objects in terms of linearizability, so that we can compose our verified transactional objects with other verified linearizable objects.

Our key technical advance is making the structure of concurrent programs more syntactic by using a *free monad* to represent syntax trees and executing them later with an explicit interpreter. We begin with *interaction trees* [Xia et al. 2020], which have demonstrated this style for sequential programs, and show how to adapt them for linearizability proofs of concurrent objects. Interestingly, in this setting, serializability (the classical transaction correctness condition [Papadimitriou 1979]) turns out to be literally a special case of linearizability for objects whose higher-order methods take code (represented as interaction trees) as arguments. Note the contrast with familiar correctness criteria for transactions, which are typically stated as ad-hoc conditions [Doherty et al. 2013; Guerraoui and Kapalka 2008; Jagannathan et al. 2005; Papadimitriou 1979; Scott 2006].

To make this framework more efficient, our library methods perform *syntactic transformations* on their interaction-tree arguments. For example, consider how *asTransaction* might transform the tree of method calls *moveStack* wishes to perform. After inlining a bit of library code, we first see insertion of code for initialization (*beginTransaction*) and finalization (*commitTransaction*, etc.).

```
moveStackAtomically(from, to) := _ ← beginTransaction;
                                 r ← moveStack′(from, to);
                                 match r with None ⇒ abortTransaction
                                            | Some(r) ⇒ _ ← commitTransaction; return(r)
```

The library builds an instrumented routine *moveStack′*, which looks just like *moveStack* but with calls to instrumented methods like *from.pop′*() instead of *from.pop*(), where *from.pop′*() is a modified version of *from.pop*() like the following,

```
pop() := ...;   (* Original code: *)          pop′() := ...;   (* Instrumented code: *)
    v ← read(i); k(v)                             vo ← trans_read(i);
                                                  match vo with None ⇒ return(None)
                                                              | Some(v) ⇒ k′(v)
```

and where the remaining method body k has been transformed similarly to yield k'. Crucially, the free monad represents programs explicitly as trees of method calls and responses to their return values, allowing us to traverse these trees syntactically and add uniform instrumentation.

Recapping, we adopt linearizable objects as the foundation of our framework. An object packages concurrent code with private state to implement public methods, and it is verified with respect to a sequential specification. The framework supports modular implementation and verification of concurrent libraries that include both classic linearizable data structures and serializable transactional objects. It formulates serializability in terms of linearizability and offers a unified proof technique where all library modules are proved linearizable against straightforward sequential specifications.

The modular nature of the framework fits our goal of verifying transactional predication. The basic idea, presented in more detail in the next section, is to combine a concurrent map with a transactional-memory library, yielding a higher-level *transactional concurrent map* abstraction.

While the map we build on provides atomicity only at the level of single-key reads and writes, the higher-level map allows grouping of several dependent reads and writes in transactions. The underlying concurrent map is used to associate keys with references to mutable memory cells managed by the transactional-memory system. As a result, key lookups have the performance of the concurrent map, while the values associated with several keys can be read or written within a single atomic transaction. Each of these main ingredients is itself constructed atop more primitive library modules with their own sequential specifications and is separately verified.

In summary, our key contributions are:

- We present a core formal framework for *verified linearizable objects* whose methods are expressed as interaction trees [Koh et al. 2019; Xia et al. 2020; Zakowski et al. 2020]. The framework includes powerful composition combinators to define implementations and specifications and verify implementations (§2 to §4), modularly. A key component of the framework is a unified proof principle for linearizability (§ 5), which we demonstrate by verifying a concurrent hash-map object (§6) and a concurrent histogram (§7).
- Within this framework, we introduce *verified transactional objects*. The key idea is to view transactions as first-class entities, represented as interaction trees, and use *interaction-tree rewriting* to instrument transactions with the bookkeeping calls required to ensure atomic execution. We use the composition operations defined earlier to state transaction serializability as an instance of linearizability. We then apply the proof principle for linearizability to prove the serializability of a transactional-memory object based on transactional mutex locks (TML) [Dalessandro et al. 2010] (§8).
- We use this framework to carry out a significant case study, leveraging its support for compositional reasoning to verify a concurrent-set object implemented using *transactional predication* (§9). To our knowledge, this correctness proof is the first rigorous one—mechanized or otherwise—for a concurrent object that encapsulates both a conventional concurrent data structure and a transactional-memory implementation.
- We package our verification framework as a *reusable Coq library* called C4 (for Certified Composable Concurrency in Coq) [Lesani et al. 2022], proved correct from first principles. C4 mechanizes everything presented in the paper, including the main case study and all its dependencies. The library is submitted as a code supplement.

We survey related work in § 11 and conclude with future directions in § 12.

2 OVERVIEW

We begin by fixing terminology and notation for some standard concepts, using the example of a simple concurrent counter; then we review the core idea of transactional predication.

Objects. We speak of *objects* encapsulating some *state* which is accessed through *methods* grouped into *interfaces*. An object implements a *high-level interface* by issuing a sequence of calls to a *low-level interface*. A core assumption, which significantly simplifies our composition laws, is that object methods may not spawn new threads—i.e., concurrency arises at the level of applications, not internally in libraries.

```
$inc := i \leftarrow r.read();$
    $ok \leftarrow r.cas(i, i+1);$
    if $ok$ then $i+1$ else $inc$
```

Fig. 2. *inc* method

Consider an object with just one method, *inc*, which increments an abstract counter and returns its new value. We can implement this object in terms of a compare-and-swap (CAS) register interface, with methods *read*, *write*, and *cas* (Figure 2). Multiple application threads may call *inc* simultaneously. For example, Figure 1(a) shows a possible execution history starting with two calls to *inc* in two concurrent threads, displaying interactions through the counter's high- and low-level interfaces. Time flows vertically. Horizontal arrows are *events*—either

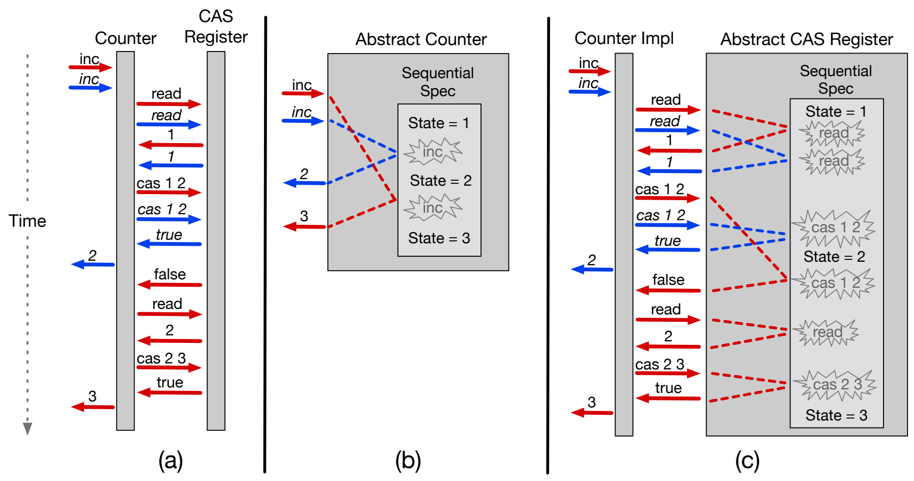


Fig. 1. (a) A possible execution history of a counter object implemented using a CAS register object. (b) A linearization of the high-level history. (c) A linearization of the low-level history.

method calls (rightward arrows) or returns (leftward arrows). Calls from the external environment to the counter object are on the left; calls from the counter to the underlying CAS register object are in the middle of the diagram. Events in thread 1 are shown in red (and normal font); events of thread 2 are blue (and italic). In this execution, both threads first execute *read* and get the same value back. Both then try to compare-and-swap, but only one of them (thread 2) succeeds, incrementing the counter once. Thread 2 returns the value of the counter, now at 2. Thread 1 tries again and finally succeeds, incrementing the counter to 3. The counter object translates single calls from its own clients (on the left) into multiple calls to its low-level interface (on the right).

Objects and linearizability. A *sequential specification* describes the high-level behavior of an object when its methods are executed sequentially, waiting for each to return before calling the next one. Formally, a sequential specification for a given interface is defined as a labeled state-transition system, where the labels are pairs of method calls and return values. Alternatively, a sequential specification can itself be viewed as an idealized "atomic object," whose methods execute fully and return their results as soon as they are called. We say that an object is *linearizable* [Herlihy and Wing 1990] with respect to a sequential specification when every method invoked on the object appears to execute atomically at some point between its call and its return, matching the behavior of the method call from the sequential specification. (We will formally define linearizability and its properties in §4.) For example, Figure 1(b) shows how the high-level history of the counter object in Figure 1(a) can also be produced by a sequence of atomic interactions with an idealized counter shown in lighter gray: the idealized counter responds first to thread 2 and then to thread 1.

So far, we have focused on one side of interfaces, where an object acts as the *callee* of its high-level interface (here, *inc*); the object must satisfy the high-level sequential specification that is associated with the interface. The object also acts as the *caller* of its low-level interface (here, *read* and *cas*), which is itself associated with a low-level sequential specification that the object can rely on in order to satisfy its high-level specification. In the example, our counter object interacts, through its low-level interface, with a CAS-register object that it assumes is linearizable with respect to its own sequential specification. Figure 1(c) shows a linearization of the low-level history from Figure 1(a).

Transactions and serializability. An appealing feature of linearizable objects is that, in every layer of a hierarchical system and its proofs, every object is proved against a straightforward sequential specification of the objects it depends on. In the example, when proving the linearizability of the counter object, we can restrict attention to histories where low-level method calls return immediately, allowing us to reason about many fewer interleavings than if we had to consider the actual implementations of *read* and *cas*. However, while the methods of a linearizable object are logically atomic, clients must still reason about possible interleavings of *sequences* of calls.

A more user-friendly model of concurrency is offered by transactional memory, which allows client programmers to choose the granularity of atomic actions, delimiting blocks of client code—possibly containing multiple calls—that must execute atomically as wholes, a requirement called serializability. We will model transactional memory's correctness as a special case of linearizability: atomic transactions can be viewed as calls to the methods of a linearizable object, where the arguments to those methods consist of *programs* to be interpreted (atomically). A program is an interaction tree that may include method calls on an interface. One familiar way to implement such an object is to interpret transactions in an environment where conflicts can be detected and transactions can be rolled back. We will formalize this approach (in §8) as a form of "transaction instrumentation," inserting "transaction life-cycle" method calls.

Transactional objects. The fine-grained conflict detection of transactional-memory protocols can hinder performance. Implementations of high-performance transactional objects [Assa et al. 2020; Bronson et al. 2010; Elizarov et al. 2019; Spiegelman et al. 2016] use TM (transactional memory) only sparingly, to reduce the frequency of "false conflicts," instead delegating most memory accesses to more efficient concurrent data structures. They achieve the best of both worlds: composability from TM and performance from concurrent data structures.

An elegant realization of this idea is the technique called *transactional predication* [Bronson et al. 2010]. Figure 3 shows the internal structure of a transactional-set object built in this style. The transactional set is vertically composed on top of the horizontal composition of two lower-level objects: a *locator* (which wraps a concurrent map) and a TM. The locator maps each element that has ever been in the set (whether or not it is still in the set at the moment) to a *location* (a mutable cell) managed by the TM. A location stores a mutable Boolean (called a *predicate*, hence the technique's name) that represents whether the element is currently in the set. The locations themselves are managed by the TM. A locator is a simple concurrent object that is built on top of a concurrent-map object. (We will later implement a concurrent-map object on top of arrays of locks and buckets and implement the TM object on top of a register and a map object.) Given an element, the locator checks whether the element is already in the map. If the element is present, the locator returns the location that the element is mapped to. If it is not, the locator puts the element and a fresh TM location in the map.

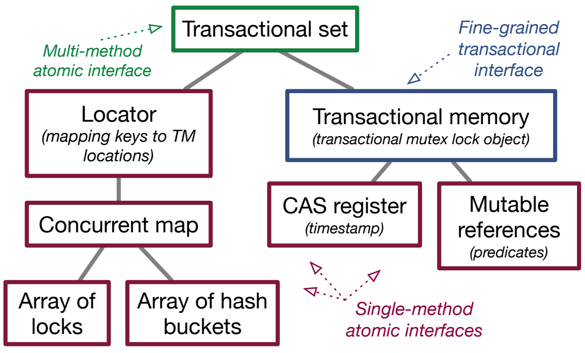

Transactional set
Multi-method atomic interface
Fine-grained transactional interface
Locator
(mapping keys to TM locations)
Transactional memory
(transactional mutex lock object)
Concurrent map
CAS register
(timestamp)
Mutable references
(predicates)
Array of locks
Array of hash buckets
Single-method atomic interfaces

Fig. 3. The architecture of transactional predication.

The interface of a transactional set accepts user programs on the set interface and executes them atomically. Given a user program, it inserts TM life-cycle calls such as TM initialization and commitment calls into the program. For each set method call on an element in the user program, the locator is called to find the location corresponding to that element. A TM read or write method on the Boolean value stored in that location is performed depending on the set method call (membership, insert, or delete). A single transaction may access several elements of the set, leading to accesses to several locations in the TM. Since the TM enforces atomicity of all such accesses to the locations, the transactional-set object inherits the same atomicity. Accesses to these locations track conflicts only at the semantic level for the set interface. Contending accesses inside the locator do not lead to conflicts. By contrast, a set implemented purely based on TM tracks conflicts on the low-level reads and writes and aborts more transactions.

Verified transactional objects. The use of a concurrent object (the locator) together with a TM raises significant challenges for verification. The TM guarantees the atomicity of transactions that use just its own interface, but the methods of the transactional set call methods on *both* the locator and the TM. How can the atomicity guarantees of the TM be used to prove the atomicity guarantees of the transactional set? An important observation is that the locator object behaves like a *pure function*, as far as its callers can tell: although the mapping from the keys to the locations is actually decided dynamically, once a mapping is made, it stays unchanged. We prove simulation relations that let us substitute method calls on the locator with ordinary function calls in the metalanguage. This substitution reduces method calls on the transactional set to transactions on the TM interface, allowing us to apply the atomicity guarantees of the TM directly. (See §9 for more detail.)

This proof style assigns each component a natural specification, without anticipating how other parts of the hierarchy will work. For instance, the specifications of classical concurrent data structures need say nothing about transactions. Also, the approach is modular: each library component can be proved separately against its natural specification.

This transactional set can be composed on top of any concurrent map and transactional memory implementing the given specifications. As concrete examples, this paper also presents concrete implementations of those specifications to illustrate the core concepts of our framework. We show a concurrent map using lock striping, which uses an array of locks to protect an array of buckets; and a transactional mutex lock (TML) object, which uses a compare-and-swap register to increment timestamps that control concurrent accesses to mutable references.

Structure. Now we are ready to dive into the details. The next section formalizes the ideas of objects, interfaces, and sequential specifications. §4 and §5 formalize linearizability, together with the related concepts of simulation and composition, and §6 and §7 apply them to verify concurrent data structures. §8 and §9 formalize transactions and transactional predication.

3 CONCURRENT OBJECTS

Interfaces. An *interface* M is a collection of *method calls*. A method call, for example $\mathsf{lookup}(k)$, intuitively consists of a method name (lookup) paired with its arguments (k)—*e.g.*, $\mathsf{lookup}(1)$ and $\mathsf{lookup}(2)$ are different method calls. The arguments to methods typically consist of first-order values (integers, strings), but they may also be *programs* (described below); indeed, this possibility will be key to our treatment of transactions in §8. Formally, we define an interface as a type $M\ R$ *indexed* by the return type R. For instance, the map interface map associating keys of type K with values of type V can be defined as follows: The type constructor map has two data constructors get and put. For any key k, there is a method call $\mathsf{get}(k) : \mathsf{map}\ (\mathsf{option}\ V)$, where the result type is option V. Similarly, for any key k and value v, there is a method call $\mathsf{put}(k, v) : \mathsf{map\ unit}$.

Programs. *Programs* are data structures that describe chains, or more generally trees, of method calls. They play two roles in our framework: as the bodies of methods associated with objects and as the bodies of transactions. We represent them as interaction trees [Koh et al. 2019; Xia et al. 2020; Zakowski et al. 2020]—intuitively, trees whose internal nodes are method calls, with one branch for each possible result.

Formally, interaction trees are defined by a datatype with three constructors. The first constructor represents a *method call* (uninterpreted, for the moment—it is just a syntactic node in a data structure). For example, the tree $x \leftarrow \mathsf{m}; p(x)$ is a program that describes making a method call m, binding the result to the variable x and continuing as the program $p(x)$, where p is a function from values to interaction trees. The second type of node, written τ, represents a *silent* step of computation. For instance, $\tau; p$ is a program that steps silently to the program p. Finally, a program may be just a *return* leaf, written v, where v is the program's result value.

This encoding of programs is important for the flexibility of the Coq framework. First, program trees are *coinductive* and hence potentially infinite, as required, for example, to implement an algorithm that retries some action until it succeeds, like the counter in §2. Silent steps are introduced by looping constructs to satisfy Coq's *productivity checker*. Second, programs are *first-class, syntactic* objects that can be manipulated by other programs, which allows encoding a variety of useful code transformations, including what we need to implement transactional memory cleanly (§8).

Implementations. An *implementation*, *impl*, of a high-level interface M in terms of a low-level interface N, written $impl : \mathsf{Impl}\, M\, N$, is a mapping $impl = (\mathsf{m} \mapsto p)$ from method calls m of M to programs p over the interface N. Formally, it is a polymorphic function ensuring that a method m is mapped to a program $impl(\mathsf{m})$ with the same result type R. The identity implementation, $\mathsf{id} : \mathsf{Impl}\, M\, M$, maps any method m to the program $(x \leftarrow \mathsf{m}; x)$, which calls the same method and returns its result. It will serve as the identity of vertical composition (in §4).

Sequential specifications. To define the semantics of an implementation, we must first define the semantics of its low-level interface by specifying the values its methods may return when called sequentially. Formally, a sequential specification *spec* of an interface M, written $spec : \mathsf{Spec}\, M$, is a labeled state-transition system: it is a tuple (S, s_0, Δ), comprising a type S of abstract states, an initial state s_0 of type S, and a relation $\Delta : \forall R, \mathcal{P}(M\, R \times S \times R \times S)$ between a method call, a current state, a result, and a next state, where $\mathcal{P}(U)$ denotes the powerset of U. (The result type R is another parameter of that relation, which we leave implicit.)

As an example, let us consider a sequential specification of locks—call it lock-spec. Its transition system has a state of type bool that represents whether the lock is acquired, with initial state false. The two methods of the interface are lock and unlock (with result type unit): the semantics of the former is to toggle the state to true, and the latter to set it to false. As another example, the sequential specification of a register, reg-spec, is a tuple of the type $\mathbb{N}$ (numbers), the initial value 0, and the obvious transition system for the methods read, write, and cas. The method cas takes two arguments, changes the value of the register to the second argument if its current value is equal to the first argument, and returns a Boolean indicating whether it succeeded. Finally, a sequential specification for maps, map-spec, can be defined as follows. The state is a mapping from keys K to values option V, with an initial state that maps any keys to the "none" value $\perp$. The interface is given by two methods (with the expected transitions): $\mathsf{get}(k)$ with the return type option V for lookup and $\mathsf{put}(k, v)$ with the return type unit for insertion.

As Figure 4 shows, a sequential specification naturally provides a stateful interpretation of programs, represented as a *transition relation* $\rightarrow_{spec}$ between pairs $(s \mid p)$ of abstract states and programs, where finished programs do not step.

$$\textsc{CallStep}\quad \frac{(\mathsf{m}, s, r, s') \in \Delta}{(s \mid x \leftarrow \mathsf{m}; p(x)) \rightarrow_{spec} (s' \mid p(r))}$$

$$\textsc{TauStep}\quad (s \mid \tau; p) \rightarrow_{spec} (s \mid p)$$

Fig. 4. Program transition

Objects. An *object obj* with interface M, written $obj : \mathsf{Object}\, M$, is a triple $(N, impl, spec)$ of low-level interface N, implementation $impl : \mathsf{Impl}\, M\, N$, and sequential $spec : \mathsf{Spec}\, N$.

Multiple threads may make method calls on the high-level interface of an object; the corresponding programs will run concurrently, interleaving calls to the object's low-level interface. The semantics of the low-level interface is given by a sequential specification, and every low-level method call operates atomically on the shared low-level state according to that specification.

The observable behavior of an object is characterized by the methods that the clients may call concurrently and the responses that they receive. Formally, behavior is modeled by the trace semantics of the following state-transition system. Let $obj = (N, impl, spec)$ be an object with high-level interface M and low-level interface N. In a concurrent execution, the state of the object consists of a pair of (1) a shared *low-level state* s of the sequential specification *spec* and (2) the states of all the threads, represented as a finite map Θ from *thread identifiers* θ (drawn from some infinite supply of names) to programs p over the low-level interface N, each representing the remainder of a method that a thread has yet to finish executing. (We write $\Theta; \theta \mapsto p$ to denote a map Θ extended with the mapping from θ to p.) The initial state of the object is the pair $(s_0 \mid \emptyset)$, containing the initial state s_0 of *spec* and the empty thread state $\emptyset$.

Spawn
$$\frac{\theta \notin \Theta}{(s \mid \Theta) \xrightarrow{\theta,\mathsf{m}}_{obj} (s \mid \Theta; \theta \mapsto impl(\mathsf{m}))}$$

Step
$$\frac{(s \mid p) \rightarrow_{spec} (s' \mid p')}{(s \mid \Theta; \theta \mapsto p) \rightarrow_{obj} (s' \mid \Theta; \theta \mapsto p')}$$

Return
$$(s \mid \Theta; \theta \mapsto v) \xrightarrow{\theta,v}_{obj} (s \mid \Theta)$$

Fig. 5. Operational semantics

The *operational semantics* of the object *obj*, shown in Figure 5, has three kinds of transitions, all written $\rightarrow_{obj}$, with different labels above the arrow.

(1) A new thread can be *spawned* to execute a high-level method call m. Such a step is represented by a transition labeled with a fresh thread identifier θ (not already in Θ) as well as the method (and arguments) m. The thread state Θ is extended with a mapping from the new identifier θ to the program that the implementation *impl* associates to the method m.

(2) A thread can step *internally*, based on the program transition relation defined above, and mutate the internal state. The environment cannot observe this transition, so it is unlabeled.

(3) A thread can *return* once it is done, *i.e.*, when the remaining program is a leaf v. This transition is labeled with the thread identifier (to identify the transition that initially spawned this thread) and its result v. The transition takes the thread out of the state. (We note that in order to issue a sequence of calls by the same thread, calls can be spawned with the same thread identifier.)

A history (or trace) is a sequence of events e—either call events (θ, m) or return events (θ, v). The *observable behavior* $\mathsf{beh}(obj)$ of an object is the set of histories produced by the above transition system ($\xrightarrow{e}_{obj}$) starting from the initial state, i.e., $\mathsf{beh}(obj) = \{(e_0 \dots e_n) \mid \exists\, s\, \Theta,\ (s_0 \mid \emptyset) \rightarrow^{\star}_{obj} \xrightarrow{e_0}_{obj} \cdots \rightarrow^{\star}_{obj} \xrightarrow{e_n}_{obj} (s \mid \Theta)\}$. Note that multiple low-level internal steps ($\rightarrow^{\star}_{obj}$, the transitive closure of $\rightarrow_{obj}$) may happen between two labeled steps, but they are not recorded in the history $(e_0 \dots e_n)$.

4 LINEARIZABILITY AND COMPOSITION

Having defined the semantics of individual objects, we now formalize the familiar notions of simulation, linearizability, and composition, their properties, and their relation with each other in a novel unified framework. We first define a notion of refinement between objects based on trace inclusion. This notion, in turn, will allow us formally to capture linearizability between an object and a specification. Then, we state properties of linearizability that support its hierarchical verification by successive refinements. Finally, we state properties of compositions that support modular verification of linearizability for composed objects.

Simulation. We define simulation relations for specifications and objects in three steps. We first define simulation between two specifications. Then, on top of that notion, we define simulation between an object and a specification when the object is executed sequentially. Finally, we define simulation between two objects when they execute concurrently.

Specification simulation. Intuitively, a specification simulates another specification with the same interface iff (1) every value that can be returned by a method call on the former can also be returned by the same method call on the latter, and (2) this property continues to hold for the resulting post-states. More precisely, we say that a specification $spec_1 = (S_1, s_1, \Delta_1)$ simulates another specification $spec_2 = (S_2, s_2, \Delta_2)$ on the same interface M, written $spec_1 \lesssim_{\mathbb{SP}} spec_2$, iff, for all m, r, and s_1', if $(\mathsf{m}, s_1, r, s_1') \in \Delta_1$, then there exists s_2' such that $(\mathsf{m}, s_2, r, s_2') \in \Delta_2$ and (S_1, s_1', Δ_1) simulates (S_2, s_2', Δ_2). This definition is *coinductive*—that is, it is defined as a greatest fixed point, allowing easy characterization of infinite executions.

Interpreted sequential specification. Any object *obj* can be interpreted in a sequential manner: the bodies of the methods are interpreted atomically as state transitions over the states of the low-level specification, by iterating the program transition relation $\rightarrow_{spec}$ until we reach a value. This convention associates *obj* to a sequential specification whose interface is the high-level interface of *obj*. More precisely, the interpreted sequential specification for the object $obj = (N, impl, spec)$ with respect to the low-level specification $spec = (S, s_0, \Delta)$ is the specification $\mathsf{interp\text{-}as\text{-}spec}(obj) = (S, s_0, \Delta')$ where, for all m, we have $\Delta' \ni (\mathsf{m}, s, v, s')$ iff $(s \mid impl(\mathsf{m})) \rightarrow^*_{spec} (s' \mid v)$.

Sequential object simulation. An object *obj* sequentially simulates a sequential specification *spec*, written $obj \lesssim_{\mathbb{S}} spec$, iff the interpreted sequential specification of *obj* simulates *spec*—*i.e.*, $\mathsf{interp\text{-}as\text{-}spec}(obj) \lesssim_{\mathbb{SP}} spec$. Intuitively, if an object sequentially simulates a specification, it behaves according to the specification when its methods are executed sequentially.

Concurrent object refinement. An object obj_1 *(concurrently) refines* another object obj_2 written $obj_1 \lesssim_{\mathbb{C}} obj_2$, if the observable behavior of obj_1 is included in that of obj_2, i.e., $obj_1 \lesssim_{\mathbb{C}} obj_2$ iff $\mathsf{beh}(obj_1) \subseteq \mathsf{beh}(obj_2)$. The objects obj_1 and obj_2 are called *concrete* and *abstract* objects, respectively. Note that concurrent refinement relates objects with a common high-level interface, while their low-level interfaces may differ: these are viewed as internal to each object.

Concurrent refinement is both reflexive and transitive, allowing verification of objects to proceed in steps. Transitivity allows us to decompose refinement proofs of an object into steps. Reflexivity allows us to refine only parts of a composite object.

Linearizability. We define linearizability of an object with respect to a specification as a concurrent refinement between the object and an "atomic object" associated with the specification.

$$
\begin{aligned}
\lesssim_{\mathbb{C}} &: \mathsf{Object}\, M \rightarrow \mathsf{Object}\, M \rightarrow \mathsf{Prop} \\
\lesssim_{\mathbb{L}} &: \mathsf{Object}\, M \rightarrow \mathsf{Spec}\, M \rightarrow \mathsf{Prop} \\
+ &: \mathsf{Object}\, M \rightarrow \mathsf{Object}\, N \rightarrow \mathsf{Object}\, (M + N) \\
\triangleright_O &: \mathsf{Impl}\, M\, N \rightarrow \mathsf{Object}\, N \rightarrow \mathsf{Object}\, M
\end{aligned}
$$

Fig. 6. Signatures of main definitions in Section 4

Atomic object. Any sequential specification *spec* of an interface M can be associated with an *atomic object* $\mathsf{atomic}(spec) = (M, \mathsf{id}, spec)$ (recall the definition of the identity implementation: $\mathsf{id} = (\mathsf{m} \mapsto (x \leftarrow \mathsf{m}; x))$). The atomic object trivially "wraps" the sequential specification as its low-level interface, delegating every method call to the corresponding method of the sequential specification. The intuition is that this object behaves atomically because every method call executes in just a single low-level step, modifying the low-level state according to the sequential specification and immediately returning a value. (Clients of such an object can still see some nondeterminism because it takes separate steps to call the method [spawn a thread], execute it, and return a value, and these steps may be interleaved with the steps of other method calls.)

This definition allows us to connect specification simulation and concurrent refinement. If one specification simulates another, then the former's atomic object refines the latter's atomic object.

LEMMA 4.1. $spec \lesssim_{\mathbb{SP}} spec' \Rightarrow \mathsf{atomic}(spec) \lesssim_{\mathbb{C}} \mathsf{atomic}(spec')$.

Linearizability. An object *obj* is *linearizable* with respect to a sequential specification *spec* iff *obj* concurrently simulates the atomic object associated with *spec*, *i.e.*, $obj \lesssim_{\mathbb{C}} \mathsf{atomic}(spec)$. We then write $obj \lesssim_{\mathbb{L}} spec$.

The proof of linearizability of an object *obj* with respect to a specification *spec* can be carried out in two steps using an intermediate specification *spec′*: first prove that *obj* is linearizable with respect to *spec′*, then prove that *spec′* simulates *spec*.

Lemma 4.2. $obj \lesssim_{\mathbb{L}} spec' \ \wedge \ spec' \lesssim_{\mathbb{SP}} spec \ \Rightarrow \ obj \lesssim_{\mathbb{L}} spec.$

Alternatively, linearizability of an object *obj* with respect to a specification *spec* can be proved hierarchically using an intermediate object *obj′*: first prove linearizability of *obj* with respect to the interpreted specification of *obj′*, then prove sequential simulation from *obj′* to *spec*. We will use this decomposition later to verify a concurrent hash-map data structure.

Lemma 4.3. $obj \lesssim_{\mathbb{L}} \mathsf{interp\text{-}as\text{-}spec}(obj') \ \wedge \ obj' \lesssim_{\mathbb{S}} spec \ \Rightarrow \ obj \lesssim_{\mathbb{L}} spec.$

Composition. Objects support two fundamental composition patterns: *horizontal composition* corresponds to the union of interfaces, while *vertical composition* interprets the low-level calls of one implementation in terms of another implementation.

Horizontal composition. Given two interfaces M_1 and M_2, we write their disjoint union as $M_1 + M_2$. We can then define the horizontal composition of implementations and sequential specifications as follows. Given two implementations $impl_1 : \mathsf{Impl}\, M_1\, N_1$ and $impl_2 : \mathsf{Impl}\, M_2\, N_2$, their horizontal composition $impl_1 + impl_2 : \mathsf{Impl}\, (M_1 + M_2)\, (N_1 + N_2)$ simply implements high-level methods from M_1 using $impl_1$ and methods from M_2 using $impl_2$. We can similarly define horizontal composition for sequential specifications $spec_1 + spec_2$ and objects $obj_1 + obj_2$. Horizontal composition satisfies the following property with respect to concurrent refinement, allowing us to refine each summand independently. Herlihy and Wing [1990] call this property the compositionality of linearizability:

Lemma 4.4. $obj_1 \lesssim_{\mathbb{C}} obj_1' \wedge obj_2 \lesssim_{\mathbb{C}} obj_2' \ \Rightarrow \ (obj_1 + obj_2) \lesssim_{\mathbb{C}} (obj_1' + obj_2').$

Reflexivity of $\lesssim_{\mathbb{C}}$ allows keeping parts of a composite object the same while others are rewritten.

Vertical composition. The vertical composition $\triangleright$ of an implementation $impl_1 : \mathsf{Impl}\, M_1\, M_2$ on top of another $impl_2 : \mathsf{Impl}\, M_2\, M_3$, written $impl_1 \triangleright impl_2 : \mathsf{Impl}\, M_1\, M_3$, is defined by interpreting the low-level calls in the body of $impl_1$ using the methods of $impl_2$. We can then define the vertical composition $\triangleright_O$ of an implementation $impl_1 : \mathsf{Impl}\, M_1\, M_2$ on top of an object obj_2 with interface M_2, which is an object $impl_1 \triangleright_O obj_2$ with interface M_1, by composing $impl_1$ with the implementation contained in obj_2.

Vertical composition satisfies the following property with respect to concurrent refinement, allowing us to refine the internal object *obj′* to *obj*:

Lemma 4.5. $obj \lesssim_{\mathbb{C}} obj' \ \Rightarrow \ (impl \triangleright_O obj) \lesssim_{\mathbb{C}} (impl \triangleright_O obj').$

It will be convenient to write the construction of an object $(N, impl, spec)$ as $impl \triangleright_S spec$, leaving implicit the low-level interface N. Then $\triangleright_O$ is defined by $impl \triangleright_O (impl' \triangleright_S spec) = (impl \triangleright impl') \triangleright_S spec$. Furthermore, because $\triangleright$, $\triangleright_O$, and $\triangleright_S$ expect different types of right operand, we can unambiguously omit parentheses in expressions involving only these three operators, and thanks to the associativity of $\triangleright$, we can even freely reassociate parentheses, modulo changing the operators so that the resulting expression is still well-typed, *e.g.*, $(f \triangleright g) \triangleright_O obj = f \triangleright_O (g \triangleright_O obj)$.

Two implementations $impl_1$ and $impl_2$ are considered equal when they map methods to the same programs, up to ignoring finite runs of silent steps τ. This notion of equality is a congruence with respect to $+$ and $\triangleright$. Lemma 4.6 summarizes equations relating $+$ and $\triangleright$ that will be needed in the proof in §9. Those equations involve the following identity elements for $+$ and $\triangleright$. As we saw before, for any interface M, there is a trivial implementation $\mathsf{id} : \mathsf{Impl}\, M\, M$ which maps a method to a program that simply calls the same method. Such a trivial implementation is an identity for vertical composition $\triangleright$. The empty interface Empty, containing no methods, is the identity for horizontal composition +, in the sense of monoidal categories: there is a left unitor, that is to say, an implementation $\mathsf{empty_L} : \mathsf{Impl}\ (\mathsf{Empty} + M)\ M$ for every interface M, satisfying various equations; there are also a right unitor and an associator, omitted for brevity.

Lemma 4.6. *Vertical and horizontal composition define a monoidal category whose objects and morphisms are respectively interfaces and implementations. In other words, they satisfy certain equations, including (among others),*

$$(1)\ (f \triangleright g) \triangleright h = f \triangleright (g \triangleright h) \qquad (3)\ (f + g) \triangleright (h + k) = (f \triangleright h) + (g \triangleright k)$$

$$(2)\ \mathsf{id} \triangleright f = f \triangleright \mathsf{id} = f \qquad (4)\ (id + g) \triangleright \mathsf{empty_L} = \mathsf{empty_L} \triangleright g$$

for any implementations f, g, h and k with interfaces that make the equations well-typed.

5 VERIFICATION OF LINEARIZABILITY

We now present a novel proof principle for concurrent refinement. It allows reasoning about the method bodies as programs (interaction trees) and does not require them to be translated to low-level labeled transition systems. Further, instead of stating and proving a simulation relation on the global states of a pool of concurrent programs, it factors and decomposes the simulation relation into separate and simpler invariants on the object state and programs, and it factors the simulation proof into separate and simpler proof obligations. The proof technique is a general method that supports verification of both lock-based and lock-free algorithms. In §6, we apply this principle and the hierarchical proof techniques that we saw in §4 to verify the linearizability of a concurrent hash-map object. We use the same principle in §7, to prove the linearizability of a concurrent histogram, and in §8, to prove the correctness of Transactional Mutex Locks (TML) [Dalessandro et al. 2010], an implementation of transactional memory.

In a concurrent execution (§3), the state of an object is a pair $(s \mid \Theta)$ of a data state s and a thread pool Θ. Stating invariants about these pairs is complicated and distracts from interesting similarities between the concrete and abstract objects themselves. Further, reasoning about the pool of threads involves boilerplate steps such as reasoning about spawning and returning threads and stating and proving conditions for every thread or pair of threads in the pool.

```
$inc' := i \leftarrow r.read();$
         $r.write(i + 1);$
         $i + 1$
```

Fig. 7. *inc′* method

Invariant relations. To prove that a concrete object refines an abstract object, we need to define three relations and prove five obligations. Start with a concrete object $(N, impl, spec)$ on the low-level specification $spec = (S, s_0, \Delta)$ and an abstract object $(N', impl', spec')$ on the low-level specification $spec' = (S', s'_0, \Delta')$. The *data relation* $\mathrm{R_D}$ captures the relation between the concrete and abstract data. The *program relation* $\mathrm{R_P}$ captures the relation between the corresponding concrete and abstract programs. Finally, the *interprogram relation* $\mathrm{R_I}$ captures the mutual relation between pairs of concrete and abstract programs.

For example, consider the *inc* method that we saw in Figure 2 of a counter object c, compared with the *inc′* method of a counter object c' in Figure 7. The concrete counter c is linearizable with respect to the interpreted specification of the abstract counter c'. Informally, the proof principle captures the following invariants. (1) The *data relation* $\mathrm{R_D}$ says that the states of the two base objects r are equal. (2) The *program relation* $\mathrm{R_P}$ captures a correspondence between the intermediate concrete and abstract programs. It is defined based on the linearization point, a low-level method call in the concrete program where the abstract program should be executed instantaneously. When the intermediate concrete program has not yet reached the linearization point, the corresponding intermediate abstract program is not executed yet. After the linearization point, the corresponding abstract program is already evaluated to a value. In this example, the linearization point is reached when the *cas* call succeeds. The entire abstract method *inc′* is executed then. (3) In this example, the *interprogram relation* $\mathrm{R_I}$ is trivially true. Our framework provides simple instantiations of the proof

$$\textsc{InitObl}:\ \mathrm{R_D}(s_0,\ s'_0)$$

$$\begin{aligned}\textsc{PPSymObl}:\ & \mathrm{R_I}(p_1,\ p_2,\ s,\ p'_1,\ p'_2,\ s') \Rightarrow \\ & \mathrm{R_I}(p_2,\ p_1,\ s,\ p'_2,\ p'_1,\ s')\end{aligned}$$

$$\begin{aligned}\textsc{CallObl}:\ & \mathrm{R_D}(s,\ s') \Rightarrow \\ & \mathrm{R_P}(\mathit{impl}(\mathsf{m}),\ s,\ \mathit{impl}'(\mathsf{m}),\ s') \\ & \wedge (\mathrm{R_P}(p,\ s,\ p',\ s') \Rightarrow \\ & \quad \mathrm{R_I}(\mathit{impl}(\mathsf{m}),\ p,\ s,\ \mathit{impl}'(\mathsf{m}),\ p',\ s'))\end{aligned}$$

$$\textsc{RetObl}:\ \mathrm{R_D}(s,\ s')\ \wedge\ \mathrm{R_P}(v,\ s,\ p',\ s')\ \Rightarrow\ p' = v$$

$$\begin{aligned}&\textsc{StepObl}: \\ &\quad \mathrm{R_D}(s_1,\ s'_1)\ \wedge \\ &\quad \mathrm{R_P}((x \leftarrow \mathsf{n}; f(x)),\ s_1,\ p'_1,\ s'_1)\ \wedge \\ &(\mathsf{n}, s_1, v, s_2) \in \Delta \Rightarrow \\ &\exists\, p'_2\ s'_2, \\ &\quad (s'_1, p'_1) \rightarrow^*_{\Delta'} (s'_2, p'_2) \\ &\quad \wedge \mathrm{R_D}(s_2,\ s'_2) && (1) \\ &\quad \wedge \mathrm{R_P}(f(v),\ s_2,\ p'_2,\ s'_2) \\ &\quad \wedge\ (\forall\, p\ p', && (2) \\ &\qquad \mathrm{R_P}(p,\ s_1,\ p',\ s'_1)\ \wedge \\ &\qquad \mathrm{R_I}((x \leftarrow \mathsf{n}; f(x)),\ p,\ s_1,\ p'_1,\ p',\ s'_1) \Rightarrow \\ &\qquad \mathrm{R_P}(p,\ s_2,\ p',\ s'_2) \\ &\qquad \wedge \mathrm{R_I}(f(v),\ p,\ s_2,\ p'_2,\ p',\ s'_2)) \\ &\quad \wedge\ (\forall\, p_1\ p'_1\ p_2\ p'_2, && (3) \\ &\qquad \mathrm{R_P}(p_1,\ s_1,\ p'_1,\ s'_1)\ \wedge \\ &\qquad \mathrm{R_P}(p_2,\ s_1,\ p'_2,\ s'_1)\ \wedge \\ &\qquad \mathrm{R_I}(p_1,\ p_2,\ s_1,\ p'_1,\ p'_2,\ s'_1) \Rightarrow \\ &\qquad \mathrm{R_I}(p_1,\ p_2,\ s_2,\ p'_1,\ p'_2,\ s'_2))\end{aligned}$$

Fig. 8. Proof principle for concurrent refinement. The concrete object is $(N, \mathit{impl}, \mathit{spec})$ on the low-level specification $\mathit{spec} = (S, s_0, \Delta)$, and the abstract object is $(N', \mathit{impl}', \mathit{spec}')$ on the low-level specification $\mathit{spec}' = (S', s'_0, \Delta')$. Unprimed and primed variables are used for the concrete and abstract objects respectively. The data relation is $\mathrm{R_D}(s, s')$, the program relation is $\mathrm{R_P}(p, s, p', s')$ and the interprogram relation is $\mathrm{R_I}(p_1, p_2, s, p'_1, p'_2, s')$. The free variables are universally quantified.

technique as well. For example, an instance does not require the specification of the interprogram relation (i.e., it is simply instantiated as true).

As another example, consider the programs in Figure 9. (To simplify the example, we show only the bodies of the methods.) On the right, we have the implementation p' of a sequential object o' with two method calls on a base object b. On the left, we have the implementation p of a concurrent object o that protects the same calls with a lock l. The concrete object o is linearizable with respect to the interpreted specification of the abstract object o'. Informally, the proof principle captures the following invariants. (1) The *data relation* $\mathrm{R_D}$ says that, when the lock l is in the released mode, the states of the two base objects b are equal. (2) The *program relation* $\mathrm{R_P}$: In this example, the linearization point is reached when the lock is released. When the intermediate concrete program has not yet reached the linearization point, the corresponding intermediate abstract program is not executed yet. However, it holds that if part of the abstract program that corresponds to the executed part of the concrete program is executed, then the state of the abstract base object will be the same as the concrete base object. When the concrete program p reaches line 4, the entire abstract program p' is executed and results in the same state for the base objects. (3) *Interprogram relation* $\mathrm{R_I}$: No two concrete programs can be in the critical section (at lines 2 and 3).

	$p :=$	$p' :=$
1	l.lock ();	
2	$b.\mathsf{m}_1()$;	$b.\mathsf{m}_1()$;
3	$x \leftarrow b.\mathsf{m}_2()$;	$x \leftarrow b.\mathsf{m}_2()$;
4	l.unlock ();	
	x	x

Fig. 9. Proof principle example

More precisely, the data relation $\mathrm{R_D}(s, s')$ defines an invariant between the low-level concrete state s and the abstract state s'. The program relation $\mathrm{R_P}(p, s, p', s')$ defines an invariant between the concrete program p paired with data state s and the corresponding abstract program p' paired with data state s'. The interprogram relation $\mathrm{R_I}(p_1, p_2, s, p'_1, p'_2, s')$ defines a mutual relation among two running concrete programs p_1 and p_2 (with data state s) and two corresponding abstract programs p'_1 and p'_2 (with data state s').

The overall simulation relation is simply defined as the conjunction of the data relation R_D between the concrete and abstract data states, the program relation R_P for each program in the thread pool (and its corresponding abstract program), and the interprogram relation R_I for each pair of programs in the thread pool (and their corresponding abstract programs).

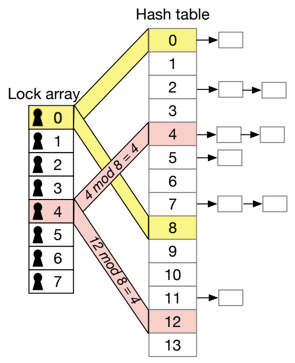
Proof obligations. Figure 8 summarizes the proof obligations. These proof obligations imply that the overall simulation relation is preserved. The obligation InitObl states that the data relation R_D holds between the initial concrete state s_0 and abstract state s'_0. PPSymObl states that the interprogram relation R_I is symmetric over programs.

The obligation CallObl states that when a method is called, the invariants are preserved. Let us consider a method m implemented by the program *impl*(m) in the concrete object and by *impl'*(m) in the abstract object. The new pair of concrete *impl*(m) and abstract *impl'*(m) programs should be in the program relation R_P with every concrete state s and abstract state s' that are in the data relation R_D. In addition, the two programs *impl*(m) and *impl'*(m) should be in the interprogram relation R_I with any pair of concrete p and abstract p' programs that are in the program relation R_P. When the concrete object returns a value, the obligation RetObl requires the abstract object to return the same value. If a concrete leaf value v is in program relation R_P with an abstract program p' and states s and s' that are also in the data relation R_D, then the abstract program p' should be the same leaf value v.

The obligation StepObl states that the invariants are preserved by program steps: when the program relation R_P holds for the pair of a concrete program $x \leftarrow \mathsf{n}; f(x)$ and an abstract program p'_1 and a pair of concrete and abstract data states s and s' that are in the data relation R_D, if the concrete program steps, then the abstract program p'_1 can take steps such that (1) both the data and program relations are preserved after the step for the resulting states and programs. In addition, it states that (2) if before the step, a pair of concrete and abstract programs p and p' were in the program relation R_P and also in the interprogram relation R_I with the concrete call program $x \leftarrow \mathsf{n}; f(x)$ and its corresponding abstract program p'_1, then after the step, the program and interprogram relations are preserved. Further, it states that (3) if before the step, two pairs of concrete and abstract programs p_1, p'_1 and p_2, p'_2 were each in the program relation R_P and also in the interprogram relation R_I with each other, then after the step, their interprogram relation is preserved with the new data states. This proof principle is for forward simulation, and the StepObl condition states proof obligations for a method call as a forward step.

```
   m(k, x) :=
L1   s ← size();
     let i := (hash k) modulo s in
L2   y ← arrayCall(i, m(k, x));
     y
```

(a) hash-map



(b) Concurrent hash-map

```
   m(k, x) :=
L1   ls ← locks.size();
     let li := (hash k) modulo ls in
L2   _ ← locks.arrayCall (li, lock);
L3   bs ← buckets.size();
     let bi := (hash k) modulo bs in
L4   y ← buckets.arrayCall (bi, m(k, x));
L5   _ ← locks.arrayCall (li, unlock);
L6   y
```

(c) conc-hash-map

Fig. 10. The implementations of sequential and concurrent hash-map objects. In (a), the metavariable m stands for either the get or put method on the map interface. The variable x is the second argument, either nothing (for get) or the value to write (for put). In (c), locks.m′() and buckets.m′() refer to methods m′ of the lock and bucket arrays respectively.

For our first example above, the three predicates of StepObl hold as follows: (1) After the step, R_D holds, because the value of the register is incremented by either both or neither of the concrete and abstract programs. Further, R_P is trivially preserved as the abstract steps are taken according to the R_P relation. (2) The R_P relation for another pair of concrete and abstract programs is preserved since R_P is independent of the data state. (3) R_I trivially holds.

In our second example: (1) After the step, R_D holds because if the lock is released, the abstract program takes a step as well, and the concrete and abstract post-states become the same. Further, R_P is trivially preserved as the abstract steps are taken according to the R_P relation. (2) When a program steps, the R_P relation for any other program is preserved as follows. The definition of R_P allows the concrete program to be in an intermediate step of the critical section only when the lock is in the acquired state. By the mutual-exclusion property of R_I, at most one of the stepping program or the other program is in the critical section. If the other program is in the critical section, the stepping program is not in the critical section and cannot release the lock to break the R_P relation for the other program. If the other program is not in the critical section, any change that the stepping program makes to the state of the lock does not affect the R_P relation for the other program. Further, the mutual-exclusion relation R_I between the two is preserved. If the other program is already in the critical section, the state of the lock is acquired, and the stepping program cannot acquire the lock and step into the critical section. (3) The relation R_I for two other programs is preserved trivially because a stepping program cannot affect the mutual-exclusion property between them.

6 HIERARCHICAL VERIFICATION OF LINEARIZABILITY

We saw the sequential specification of maps, map-spec, in §3. We next implement both sequential and concurrent hash-map objects, hash-map and conc-hash-map. The former uses just an array of buckets; the latter, in addition, uses an array of locks. We show that the concurrent hash-map object conc-hash-map is linearizable with respect to the sequential specification of maps map-spec. The proof is divided into two steps, by Lemma 4.3, with the sequential object hash-map as an intermediate specification: we first show that hash-map sequentially simulates map-spec, and then we show that conc-hash-map is linearizable with respect to the interpreted sequential specification of hash-map.

Sequential hash-map. The sequential hash-map object hash-map is implemented as a vertical composition on top of an array object, which represents an indexed sequence of other objects. (Formally, the array object type is parametric—at the metalevel—in the cell object type: for each type of cells, there is a separate type of arrays of those cells.) The array interface provides two methods: a method arrayCall that, given an index and a method call, calls the method on the object stored at that index; and a method size that returns the size of the array. Each cell of the array refines the object that the array is instantiated with.

We use a closed-address hash-map, where each bucket refers to a set of items. In the hash-map object, the array elements are map objects that represent buckets with colliding keys. The bucket map bmap can be any simple map object (thus, the map interface plays both high- and low-level roles in the implementation of the hash-map). Figure 10a presents the implementation of methods of the hash-map. Map methods are parametrized by the key k being accessed, which we hash to obtain the index i of the corresponding bucket. We then call the input method on the map object contained in that bucket and return its result.

If the given bucket-map object sequentially simulates the map specification, the hash-map object sequentially simulates the map specification as well:

Lemma 6.1. bmap $\lesssim_{\mathbb{S}}$ map-spec $\Rightarrow$ hash-map $\lesssim_{\mathbb{S}}$ map-spec.

Concurrent hash-map. We implement a striped concurrent hash-map as an object conc-hash-map. The structure of the buckets is similar to the sequential hash-map that we saw above. The bucket map bmap is not linearizable (not thread-safe). As Figure 10b shows, the implementation uses lock striping to protect access to buckets. An array of locks is used, and the lock at index i protects all the buckets at indices j such that j is i modulo the size of the lock array. Thus, the concurrent hash-map object is a vertical composition on the horizontal composition of two objects: an array of bucket maps (bmap) and an array of locks (lock). Figure 10c presents the implementation of the methods of the concurrent hash-map. It gets the size of the lock array, gets the input key of the call, computes the lock index as the hash value of the input key modulo that size, acquires the lock at the lock index, then gets the size of the bucket array, computes the bucket index as the hash value of the input key modulo that size, performs the input method on the bucket index, releases the lock at the lock index, and returns the resulting value.

To prove the correctness of the concurrent hash-map, we use the sequential hash-map as an intermediate specification. The following lemma states that the concurrent hash-map object is linearizable with respect to the interpreted specification of the sequential hash-map.

Lemma 6.2. $\mathsf{lock} \lesssim_{\mathbb{L}} \mathsf{lock\text{-}spec} \Rightarrow \mathsf{conc\text{-}hash\text{-}map} \lesssim_{\mathbb{L}} \mathsf{interp\text{-}as\text{-}spec}(\mathsf{hash\text{-}map})$

Finally, we apply the hierarchical technique of Lemma 4.3 to Lemmas 6.2 and 6.1. We conclude that the concurrent hash-map object is linearizable with respect to the map specification.

Corollary 6.3. $\mathsf{bmap} \lesssim_{\mathbb{S}} \mathsf{map\text{-}spec} \wedge \mathsf{lock} \lesssim_{\mathbb{L}} \mathsf{lock\text{-}spec} \Rightarrow \mathsf{conc\text{-}hash\text{-}map} \lesssim_{\mathbb{L}} \mathsf{map\text{-}spec}$

7 LINEARIZABILITY OF VERTICAL COMPOSITIONS

Linearizability ensures that concurrent method calls on the object appear to execute atomically and behave according to the sequential specification of the object. This guarantee is only provided for individual method calls on the object. However, methods of a vertical composition on top of the object may make multiple calls to the object; therefore, they are not necessarily atomic. Studies of production code [Shacham et al. 2011] shows that atomicity bugs are prevalent in vertical compositions. In this section, we see how a concurrent histogram can be implemented as a vertical composition on top of a concurrent hash-map and how its linearizability can be verified using the same proof technique that we saw in §5.

```
      hist-imp :=
        get (k) :=
$G_1$     $r \leftarrow$ map.get (k);
$G_2$     r

        inc (k) :=
$I_1$     $r \leftarrow$ map.get (k);
$I_2$     match r with
$I_3$     | $\lceil v \rceil \Rightarrow$
$I_4$       $s \leftarrow$ map.replace $(k, v, v+1)$;
$I_5$       if (s)
$I_6$         $v + 1$
$I_7$       else
$I_8$         inc (k)
$I_9$     | $\perp \Rightarrow$
$I_{10}$      $r \leftarrow$ map.putIfAbsent(k, 1);
$I_{11}$      match r with
$I_{12}$      | $\lceil \_ \rceil \Rightarrow$ inc (k)
$I_{13}$      | $\perp \Rightarrow 1$
```

Fig. 11. The implementation of the Histogram object on a Map object

A histogram is a data structure that represents values for a set of bars. We first consider the sequential specification of histograms hist-spec. The interface of a histogram is parametric in terms of the key type K for the bars, and it provides two methods: get(k) and inc(k). The state is a mapping m from keys K to optional natural values option $\mathbb{N}$. The transition system of hist-spec makes a transition on get(k) that keeps the state the same and returns the value of k in the current state m. It makes two different transitions on inc(k) depending on whether k is in the domain of the current state map m. If k already exists, it increments its value in m; otherwise, it adds to the map m a mapping from k to the "some" value of 1. In both transitions, it returns the value of k in the post-state.

We will see a histogram object that is implemented by a vertical composition on top of a map object. This map object provides an extended interface: in addition to the methods get(k) and put(k, v), it provides putIfAbsent(k, v) and replace(k, v, v'). Similar to the basic map, the sequential specification of the extended map emap-spec stores a mapping m from keys to optional values. Its transition system makes two different transitions on $\mathsf{putIfAbsent}(k, v)$: if k is not in the domain of the state map m, it extends m with a mapping from k to v; otherwise, it leaves m the same. In both cases, it returns the optional value of k in the prestate. Similarly, there are two possible transitions on $\mathsf{replace}(k, v, v')$: if the value of m for k is v, it replaces v with v' and returns true; otherwise, it leaves the state unchanged and returns false.

The implementation hist-imp of the histogram object is presented in Figure 11. The get method is simply delegated to the underlying map object (at G_1). On the other hand, a naive implementation of the inc method, which would simply get the current value and put back an incremented value, is not atomic, and concurrent calls on it can easily violate the sequential specification. In the implementation of inc, the current value of the key k is first obtained from the underlying map object (at I_1). A "some" value containing an actual value v is represented as $\lceil v \rceil$, and the "none" value is represented as $\perp$. If the key k is already mapped to some value v (at I_3), it should be updated to $v + 1$. In order to avoid racing updates to the underlying map, the inc method attempts to replace v atomically with $v + 1$ (at I_4). If the value of k is not changed between I_1 and I_4 by a concurrent update, the replace call succeeds and returns true. In this case, the increment is performed and the new value $v + 1$ is returned (at I_6). Otherwise, the inc method is repeated by a recursive call (at I_8). (We note that the inc method has a coinductive definition.) If the key k is not in the underlying map (at I_9), a new mapping from k to 1 needs to be added. Similarly to the previous case, in order to avoid racing updates to the underlying map, the inc method attempts to put the value 1 for k atomically via a putIfAbsent call (at I_{10}). The putIfAbsent method always returns the previous value of k. Therefore, if it fails, it returns some value (at I_{12}), and the inc method is called again. If it succeeds, it returns none $\perp$ (at I_{13}), and the new value 1 is returned.

The vertical composition of the histogram implementation on top of the extended map is linearizable with respect to the sequential specification of the histogram.

Lemma 7.1. hist-imp $\triangleright_O$ emap-spec $\lesssim_{\mathbb{L}}$ hist-spec.

The proof of this lemma uses the proof technique that we saw in §5 and used in §6. The invariants for this proof are the following: (1) The *data relation* $\mathrm{R_D}$ says that the states of the underlying concrete map and the abstract map of the histogram are equal. (2) The *program relation* $\mathrm{R_P}$ captures a correspondence between the intermediate concrete and abstract programs according to the linearization points. Similar to the previous use cases, when the intermediate concrete program has not yet reached the linearization point, the corresponding intermediate abstract program is not executed yet. After the linearization point, the corresponding abstract program is already evaluated to a value. The linearization point of the get method is the get call on the underlying map (at G_1). The linearization point of the inc method is its last replace (at I_4) or putIfAbsent (at I_{10}) method call before its returns. (3) The *interprogram relation* $\mathrm{R_I}$ is trivially true.

8 TRANSACTIONS

Linearizability offers simple guarantees at the level of individual method calls, but it is still up to the caller to compose these guarantees to reason about complex programs that successively call multiple methods. A more convenient model for concurrent programming is offered by *transactions*: arbitrary programs, combining multiple calls to some interface, which are expected to run atomically as a whole. In this section, we first characterize the specification of transactions and then present objects that implement the specification.

Strict serializability. Given an interface, a transaction is a program p on this interface. A transaction is expected to execute atomically: either execute completely or abort without any effect. To embody this idea, we introduce a special interface whose one method ?exec takes a whole program as an argument, represented as an interaction tree. This method is expected to run the transaction atomically. We equip ?exec with a sequential specification by lifting the sequential specification of the underlying interface used by the transaction, as we describe next.

Program specification. The program specification corresponding to a given specification $spec = (S, s_0, \Delta)$ for the interface M, written prog-spec($spec$), is the specification (S, s_0, Δ'), which describes an interface Prog M with just one method, exec(p), where p is itself a program on M. The relation Δ' is defined as $(\text{exec}(p), s, v, s') \in \Delta'$ if and only if $(s,\ p) \rightarrow^*_{spec} (s',\ v)$, which says that when exec(p) runs starting from the initial state s, it ends in a state s' obtained by interpreting the sequence of method calls in p based on $spec$.

Abortable specifications. Transactions execute concurrently and may conflict on their accesses to the shared state. Therefore, atomic execution of one transaction may prevent atomic execution of another. Thus, transactions may be aborted, and the client can either retry immediately or back off to reduce contention and retry. To capture this behavior, we define abortable specifications.

$$\frac{(\mathsf{m}, s, r, s') \in \Delta}{(?\mathsf{m}, s, \lceil r \rceil, s') \in \Delta'} \qquad (?\mathsf{m}, s, \bot, s) \in \Delta'$$

Fig. 12. Abortable transition relation

First, an interface M can be translated into an *abortable interface* $?M$. For any method $\mathsf{m} : M\,R$, there is a corresponding *abortable method* $?\mathsf{m} : ?M\ (\{\bot\}+R)$, where the result of the method is made optional: it is either "some" value $\lceil r \rceil$ containing an actual result r, or it is the "none" value $\bot$.

The abortable version of a given specification (S, s_0, Δ) over the interface $M = \overline{\mathsf{m}}$, written ab-spec($spec$), is then a specification (S, s_0, Δ') over the abortable interface $?M = \overline{?\mathsf{m}}$. The transition relation Δ', shown in Figure 12, can nondeterministically either execute the method call and return its result $\lceil r \rceil$ or else abort without changing the state and return the "none" value $\bot$.

Transactional specification. Based on the above definitions of program and abortable specifications, we can now define a transactional specification, which characterizes the atomic execution of full programs, where aborting is always a possibility. The transactional specification of a given specification $spec$, written trans-spec($spec$), is simply: ab-spec(prog-spec($spec$)).

Strict serializability. An object obj that implements interface M is strictly serializable with respect to a specification $spec$ of M, written $obj \lesssim_{\mathbb{SS}} spec$, if and only if obj is linearizable with respect to the transactional specification of $spec$, i.e., $obj \lesssim_{\mathbb{L}}$ trans-spec($spec$).

$$\begin{aligned} \text{prog-spec} &: \text{Spec}\,M \rightarrow \text{Spec}\,(\text{Prog}\,M) \\ \text{ab-spec} &: \text{Spec}\,M \rightarrow \text{Spec}\,(?M) \\ \text{trans-spec} &: \text{Spec}\,M \rightarrow \text{Spec}\,(?\text{Prog}\,M) \end{aligned}$$

Fig. 13. Program, abortable, and transactional specification transformers

There is a close relation between linearizability and strict serializability. Herlihy and Wing [1990] mention that "linearizability can be viewed a special case of strict serializability where transactions are restricted to consist of a single operation." Our modular framework captures strict serializability as an instance of linearizability with "operations" (*i.e.*, methods) as programs.

Transactional objects. Having defined strict serializability, we now turn our attention to constructing verified transactional objects—objects that provably meet the specification trans-spec($spec$) and therefore provide ?exec methods, which take as input transactions (programs) and run them atomically. We define a two-piece template for doing so. First, we *instrument* the input program, inserting "life-cycle" methods to initiate, commit, and abort transactions. Second, we implement the methods of those instrumented transactions in an object, called a *transaction protocol object.* Below, we first consider the interface of a transaction protocol and the corresponding instrumentation. Then we construct the transactional object as a vertical composition of the instrumented program on top of a transaction protocol object.

Note that this construction is a reusable instance of the compositional definition and verification methodology that we saw in §4. The instrumentation function is independent of the transaction-protocol object. Therefore, the user programs can be instrumented independently and can be composed with different underlying protocols that have different performance characteristics. In addition, decoupling the protocols allows them to be used as the underlying objects of other transformations, as we will see for predicated objects in §9.

Transaction protocol interface. A transaction protocol interface trans(M) wraps another interface M, making it so that the interface M can be used inside of a transaction. For instance, we can obtain the interface for transactional memory by wrapping the map interface into a transactional protocol interface. A transaction protocol interface trans(M) provides the following four methods: init, lift, abort, and commit, which define the life-cycle methods of a transaction.

The method init: TLocal initializes the transaction and returns the transaction-local state (for example, an initial timestamp for the transaction). This state is passed between and updated by the other three methods during execution. The method lift: $\mathsf{TLocal} \times M\,R \rightarrow (\mathsf{TLocal} \times \mathsf{Option}\,R)$ takes the current transaction-local state and a method call m and tries to execute m. It returns a pair including the updated transaction-local state. The execution of m may not be successful, due to a conflict, so lift also returns an optional value, where $\bot$ indicates failure, and $\lceil r \rceil$ indicates that m successfully returned r. If a call is not successful, then the method abort: $\mathsf{TLocal} \rightarrow \mathsf{unit}$ is subsequently called to clean up before the transaction is aborted. Finally, the method commit: $\mathsf{TLocal} \rightarrow \mathsf{Option}\,\mathsf{TLocal}$ commits the transaction. Committing may itself fail due to a conflict, so it returns an option value. If it is *not* successful, commit returns the transaction-local state to be passed to the subsequent abort call; otherwise, when successful, it returns $\bot$.

```
instrument(?exec(p)) :=
    t ← init ();
    instrument′(t, p)

instrument′(t, p) :=
    match p with
    | y ← m(x); f(y) ⇒
        r ← lift(t, m(x));
        instRet(f, r)
    | r ⇒
        o ← commit(t);
        instEnd(r, o)
    | τ; p ⇒
        τ; instrument′(t, p)

instRet(f, r) :=
    let ⟨t, o⟩ := r in
    match o with
    | ⌈r⌉ ⇒
        τ; instrument′(t, f(r))
    | ⊥ ⇒
        abort t;
        ⊥
instEnd(r, o) :=
    match o with
    | ⌈t⌉ ⇒
        abort t;
        ⊥
    | ⊥ ⇒ ⌈r⌉
```

Fig. 14. Transaction instrumentation

Transaction instrumentation. Given a transaction protocol interface trans(M) and a program p that uses M, the function instrument presented in Figure 14 transforms p into a program over trans(M). (1) The method init is inserted at the beginning of the transaction. (2) Each method m in the program is executed by the lift method. The execution may succeed, in which case the continuation is instrumented, or it may fail, in which case abort is called. This logic is handled by instRet. (3) Finally, if the program returns r, commit is called. Since that might fail, instEnd checks the outcome and triggers an abort, returning $\bot$, upon failure; otherwise the value resulting from the execution of the program is returned.

Transactional objects. Instrumented transactions need to run on top of some other object that provides life-cycle methods. Given an implementation *pro* of the transaction protocol interface trans(M), we obtain a transactional object by instrumentation and vertical composition: trans-obj(*pro*) = instrument $\triangleright_O$ *pro*. The protocol object *pro* is correct if the transactional object it produces is strictly serializable for any set of programs. That is, with respect to the specification *spec*, we have trans-obj(*pro*) $\lesssim_{\mathbb{SS}}$ *spec*. The following lemma formalizes a way of decomposing that obligation, for an arbitrary candidate object.

Lemma 8.1. *A candidate protocol object $tm = (M_0, tm\text{-}imp, spec_0)$ is strictly serializable with respect to* map-spec *iff* (instrument $\triangleright$ *tm-imp*) $\triangleright_S$ $spec_0$ $\lesssim_{\mathbb{L}}$ trans-spec(map-spec).

Transactional Mutex Locking. It remains to see how to implement a transaction protocol object correctly. For the case of the map interface (which yields an implementation of transactional memory), we proved two implementations are strictly serializable: the single global lock [Menon et al. 2008] protocol and the Transactional Mutex Locking (TML) protocol [Dalessandro et al. 2010]. We illustrate the technique with TML.

The TML protocol object tml is a transaction protocol object for the map interface. It provides get and put methods on a map of keys to values. The protocol object uses a pair of a register and a map as its low-level interface. The register represents the global clock for the protocol, and the map stores values for the keys. We say that a transaction is a writer if it executes at least one put method and is a reader if it executes no put methods. Writer transactions acquire exclusive access to the map by compare-and-swapping the parity of the global clock from even to odd, and they later release it by changing the parity of the global clock back to even. Reader transactions execute optimistically: they do not acquire any locks and do not block writer transactions.

```
     init () :=
I1     t ← reg.read ();
       if (t mod 2 = 1) init ()
       else t

     lift (t, m′) :=
       match m′with
         | get (k) ⇒
G1          v ← map.get (k);
G2          gt ← reg.read ();
            if (gt = t)
              ⟨t, ⌈v⌉⟩
            else ⟨t, ⊥⟩
         | put (k, v) ⇒
            if (t mod 2 = 0)
P1            b ← reg.cas (t, t + 1);
              if (b)
P2              _ ← map.put (k, v);
                ⟨t + 1, ⌈unit⌉⟩
              else ⟨t + 1, ⊥⟩
            else
P3            _ ← map.put(k, v);
              ⟨t, ⌈unit⌉⟩

     abort (t) := unit

     commit (t) :=
       if (t mod 2 = 1)
C1       _ ← reg.write(t + 1);
         ⊥
       else ⊥
```

Fig. 15. The implementation of the TML protocol tml-imp. reg.m() is a method call to the register interface, and map.m() is a call to the map interface

Figure 15 represents the implementation of the TML protocol object. In the recursive init method, the transaction reads the global clock (I_1) in a loop until it is *even* (i.e., there is no concurrent writer transaction). The transaction keeps the final time that it reads in init as its transaction-local time t.

To lift the get method, first the value of the input key is read from the map (G_1), and then the global clock is read (G_2). If the global time is still equal to the transaction-local time, no writer transaction has been active in the meantime, and the values that the current transaction has read are still valid. Hence, the transaction returns the read value. Otherwise, the values that the transaction has speculatively read may have changed, and it returns failure.

For the put method, it is first checked if the current transaction has already acquired exclusive access to the map, i.e., the transaction-local time is odd. If that is the case, the transaction can access the map directly (P_3). Otherwise, the transaction tries to acquire exclusive access by incrementing the global time using a compare-and-swap (P_1). If the compare-and-swap is not successful, the global clock has changed since it has been read in the init method. Another writer transaction has acquired exclusive access to the map and may have changed it. Therefore, the values that the current transaction may have read may not be not valid anymore; thus, the method returns failure. If the compare-and-swap is successful, then the map is updated with the input key and value (P_2). The transaction keeps the fact that it has acquired exclusive access by incrementing its transaction-local time to the next odd value.

TML performs updates only if the transaction already has exclusive access and it is safe to perform the updates. Therefore, the abort method simply returns. The commit method checks the parity of the transaction-local time. If the transaction-local time is odd, the current transaction is a writer and has acquired exclusive access. It releases the exclusive-access right by incrementing the global time to the next even value (C_1). Finally, the commit method returns successfully.

Lemma 8.2. *The TML protocol object* tml *is strictly serializable with respect to the map interface.*

By Theorem 8.1 and the definition of transactional specification, we should prove the following concurrent refinement,

$$(\mathsf{instrument} \triangleright \mathsf{tml\text{-}imp}) \triangleright_S (\mathsf{reg\text{-}spec} + \mathsf{map\text{-}spec}) \lesssim_{\mathbb{C}} \mathsf{atomic}(\mathsf{ab\text{-}spec}(\mathsf{prog\text{-}spec}(\mathsf{map\text{-}spec})))$$

Intuitively, any step of a user program that is instrumented and vertically composed on tml should be refined by the whole execution of the user program or by not running it at all. To prove this refinement, we use the same proof principle that we saw in Figure 8. The unfixed linearization point of a reader transaction is the method call G_2 of its last successful get method call. Otherwise, it is C_1 (for a writer transaction). An intuitive explanation of the linearization points and the three invariant relations is available in the submitted supplementary material.

9 TRANSACTIONAL PREDICATION

We show an example of composing a linearizable object and a serializable object: transactional predication. Whereas, in §6, we implemented a linearizable map object, which guarantees the atomicity of individual map methods, here we want to implement a serializable map object, which guarantees the atomicity of arbitrary map transactions, *i.e.*, programs that may involve many map method calls. Transactional predication is a technique to implement a serializable map object given a linearizable map (§6) and a TM, *i.e.*, a serializable object of mutable references (§8). The general idea is that the linearizable map quickly maps keys to references, so that high-level calls can be transformed into calls on references managed by the TM. The TM is used only for the final critical calls, which avoids expensive repeated TM calls but gains TM's composability properties.

Transactional predication may be a simple idea, but formalizing it is challenging because it implements a serializable object using linearizable and serializable objects. A naive and tedious verification approach would be to reason about how arbitrary high-level transactions are reduced to low-level transactions. Instead, we define transactional predication as a composition of implementations, and we reason about it compositionally and algebraically, by equational reasoning about vertical and horizontal object compositions ($\triangleright$, $+$).

Transactional map. We saw the structure of the predicated set in Figure 3. The predicated map is similar except that instead of a Boolean, a location stores the value of the key or $\bot$ if the key is removed. The predicated map implements the high-level map interface (from §3) from keys Key to values Val using the sum RefMap + TM of the two low-level interfaces RefMap and TM. The interface TM is the map interface (that we saw in §3) instantiated with Ref as the keys and Val as values. The references are abstract locations (users may not inspect them, and they are typically meant to be references to mutable memory cells). This interface is implemented by a serializable object such as TML (§8), *i.e.*, programs consisting of get and put calls can be executed atomically.

Locator. The interface RefMap is a map of keys Key to references Ref, with only one method lookup : Key $\longrightarrow$ Ref. It is implemented by the locator shown in Figure 16, atop a low-level map which provides methods $\mathsf{get}(k)$ and $\mathsf{putIfAbsent}(k, v)$. The locator object is a "lazy" map from keys to mutable references (locations) managed by the underlying transactional memory. When a key is looked up for the first time, a new reference will be allocated and returned; when the same key is looked up again, the same reference will be returned.

The locator object should appear to behave as a pure function ϕ : Key $\longrightarrow$ Ref assocating map keys to references. A subtlety is that references will be allocated dynamically, so the function ϕ is only determined at runtime. In order to formally relate ϕ to the locator object, we convert ϕ

into a sequential specification ref-map-spec$_\phi$ parametrized by the pure function ϕ : Key $\to$ Ref, whose transitions associate lookup(k) calls to return values $\phi(k)$. Since the return value is entirely determined by the method call, the state type is trivially a singleton. The correctness theorem for a locator does not exactly match the general definition of linearizability, because the abstract object ref-map-spec$_\phi$ now depends on the trace of the concrete object locator. For any history h of the object locator ($h \in$ beh(locator)), there exists a function ϕ : Key $\to$ Ref such that h is a history of the specification ref-map-spec$_\phi$, i.e., beh(locator) $\subseteq \cup_\phi$ beh(ref-map-spec$_\phi$).

In interest of space, we elide the details of the application of that theorem. The rest of the refinement proof in the remainder of this section is after refining the locator to its specification. Thus, that proof will be applied to the history h from which we obtained the function ϕ.

```
locator (lookup($k$)) :=
  $o \leftarrow$ get ($k$);
  match $o$ with
  | $\lceil p \rceil \Rightarrow p$
  | $\bot \Rightarrow$
    $p \leftarrow$ newRef;
    $p \leftarrow$ putIfAbsent ($k, p$);
    $p$
```

Fig. 16. Locator implementation

Given a function ϕ, we can also define another implementation pure-map$_\phi$ = (lookup$(k) \mapsto \phi(k)$) of the specification ref-map-spec$_\phi$, by directly using ϕ to answer lookup calls. This alternative representation of ref-map-spec$_\phi$ is useful in equational proofs since it abstracts away the "lazy" implementation. The low-level interface of pure-map$_\phi$ is Empty as the object simply returns the value of ϕ for k without making any low-level calls. The specification ref-map-spec$_\phi$ and the implementation pure-map$_\phi$ are related by the following equality (which we will use in later proofs).

Lemma 9.1. *For all ϕ and s,* id $\triangleright_S$ (ref-map-spec$_\phi + s$) $=_\mathbb{C}$ (pure-map$_\phi$ + id) $\triangleright$ empty$_\mathsf{L}$ $\triangleright_S$ s *where $obj_1 =_\mathbb{C} obj_2$ means $obj_1 \lesssim_\mathbb{C} obj_2 \wedge obj_2 \lesssim_\mathbb{C} obj_1$.*

The key idea is that a program that makes calls to the interface RefMap + M intuitively has the same behavior as the program with interface M obtained by interpreting away the RefMap calls using ϕ. The pure-map$_\phi$ object does not use a low-level interface, so pure-map$_\phi$ + id can be composed on top of empty$_\mathsf{L}$ to filter calls on its left low-level interface. The resulting programs call methods only on the interface M (of s).

```
pred-map ($m$) :=
  match $m$ with
  | get ($k$) $\Rightarrow$
    $r \leftarrow$ RefMap.lookup ($k$);
    $v \leftarrow$ TM.get ($r$);
    $v$
  | put ($k, v$) $\Rightarrow$
    $r \leftarrow$ RefMap.lookup ($k$);
    TM.put ($r, v$);
    $\bot$
```

Fig. 17. Transactional predication. RefMap.m() is a method call to the RefMap interface, and TM.m() is a method call to TM.

Predication. The implementation of predicated map pred-map is shown in Fig. 17. The function pred-map interprets a map method m that is parametrized by the key k. It first looks up the reference r associated with the key k by a lookup in the RefMap interface. Then, it accesses the value stored in the reference via get or put in the TM interface. The implementation pred-map is used later to instrument transactions over the exposed map interface.

The predicated map reduces calls on its map interface to calls on another map. Thus, it is straightforward that the object with the implementation pred-map on the low-level specification ref-map-spec$_\phi$ + map-spec *sequentially* simulates the high-level specification map-spec.

Lemma 9.2. interp-as-spec (pred-map $\triangleright_S$ (ref-map-spec$_\phi$ + map-spec)) $\lesssim_\mathbb{SP}$ map-spec .

Instrumentation. To instrument a transaction p, we first compose it on top of the pred-map implementation to obtain a program over the sum interface RefMap + TM. Then, we instrument the program with the function instrumentR, a variant of instrument, defined in Figure 14, where the main change is that instrumentR only lifts method calls m that belong to the TM interface, while forwarding RefMap calls without modification. We then compose it on top of id + *tm-imp* to interpret the TM calls with the TM implementation *tm-imp*.

$$\text{p-map-imp} = \text{transF}(\text{pred-map}) \triangleright \text{instrumentR} \triangleright (\text{id} + \textit{tm-imp})$$

The operator transF, defined below, transforms an implementation f of an interface M into an implementation $\text{transF}(f)$ of the interface $?\text{Prog}(M)$. Given a transaction p, $\text{transF}(f)$ interprets p using f and makes a single call using the resulting transaction $p \triangleright f$.

$$\text{transF} : \text{Impl}\, M\, N \longrightarrow \text{Impl}\, (?\text{Prog}(M))\, (?\text{Prog}(N))$$
$$\text{transF}(f) = (?\text{exec}(p) \mapsto (v \leftarrow ?\text{exec}(p \triangleright f); v))$$

The following lemma states the relation between instrument and instrumentR.

Lemma 9.3. *For all p, f and g,*

$$\text{transF}(f) \triangleright \text{instrumentR} \triangleright (g + \text{id}) \triangleright \text{empty}_\text{L} = \text{transF}(f \triangleright (g + \text{id}) \triangleright \text{empty}_\text{L}) \triangleright \text{instrument}.$$

The function instrumentR is applied to programs on a sum interface $M + \text{TM}$, and it wraps calls on the right but does not alter calls on the left. Therefore, the methods on the left can be interpreted using an implementation g before the instrumentation. Further, since the composition on empty_L removes the calls on the left, the function instrument can be applied instead of instrumentR.

Strict serializability. The final transactional object is

$$\text{p-map} = \text{p-map-imp} \triangleright_S (\text{ref-map-spec}_\phi + \textit{spec}_0)$$

where $\textit{spec}_0$ is the specification of the low-level interface of the TM protocol. The goal is to prove the serializability of this object with respect to the map specification map-spec.

Before the proof, we give a helper lemma. It states that interpreting a program p by an implementation f and then interpreting it under the transactional specification of s is the same as interpreting it directly as a transaction under the interpreted sequential specification of f.

Lemma 9.4. *For all f and s,* $\text{transF}(f) \triangleright_S \text{trans-spec}(s) \lesssim_{\mathbb{SS}} \text{interp-as-spec}(f \triangleright_S s)$.

The proof of serializability is equational, *i.e.*, by successive refinements. This proof style allows us to package the correctness conditions of individual components as (in)equations that are then chained together in the final proof in well-delimited rewriting steps interleaved with some administrative simplifications or factorizations provided by the equational theory of interaction trees (Lemma 4.6). The high-level idea is that the pure calls to ref-map-spec_ϕ on the left can be filtered (step 2), and programs that are instrumented on the right by instrumentR can be transformed to flat programs that are instrumented by instrument (step 5). Having isolated the TM implementation *tm-imp*, its serializability guarantees can be applied (step 7). Finally, the correctness theorem of the core implementation exposes the map sequential specification (step 11). Those key steps rely respectively on properties of the low-level specification ref-map-spec, the instrumentation functions instrument and instrumentR, the serializable object *tm*, and the core implementation pred-map laid out above.

(1) $\text{p-map} = \text{p-map-imp} \triangleright_S (\text{ref-map-spec}_\phi + \textit{spec}_0)$
(2) By Lemma 9.1 (after expanding with $\text{p-map-imp} = \text{p-map-imp} \triangleright \text{id}$):
$\lesssim_\mathbb{C} \text{p-map-imp} \triangleright (\text{pure-map}_\phi + \text{id}) \triangleright \text{empty}_\text{L} \triangleright_S \textit{spec}_0$
(3) Unfold p-map-imp:
$\text{transF}(\text{pred-map}) \triangleright \text{instrumentR} \triangleright (\text{id} + \textit{tm-imp}) \triangleright (\text{pure-map}_\phi + \text{id}) \triangleright \text{empty}_\text{L} \triangleright_S \textit{spec}_0$
(4) Commute vertical compositions based on properties of $\triangleright$ and $+$ (Lemma 4.6):
$= \text{transF}(\text{pred-map}) \triangleright \text{instrumentR} \triangleright (\text{pure-map}_\phi + \text{id}) \triangleright (\text{id} + \textit{tm-imp}) \triangleright \text{empty}_\text{L} \triangleright_S \textit{spec}_0$
$= \text{transF}(\text{pred-map}) \triangleright \text{instrumentR} \triangleright (\text{pure-map}_\phi + \text{id}) \triangleright \text{empty}_\text{L} \triangleright \textit{tm-imp} \triangleright_S \textit{spec}_0$

(5) By Lemma 9.3:
$= \text{transF}(\text{pred-map} \triangleright (\text{pure-map}_\phi + \text{id}) \triangleright \text{empty}_\text{L}) \triangleright \text{instrument} \triangleright \textit{tm-imp} \triangleright_S \textit{spec}_0$
(6) Abbreviate $m = \text{pred-map} \triangleright (\text{pure-map}_\phi + \text{id}) \triangleright \text{empty}_\text{L}$:
$= \text{transF}(m) \triangleright \text{instrument} \triangleright \textit{tm-imp} \triangleright_S \textit{spec}_0$
(7) By serializability of *tm* (Lemma 8.1):
$\lesssim_\mathbb{C} \text{transF}(m) \triangleright_O \text{atomic}(\text{trans-spec}(\text{map-spec}))$
(8) By definition of atomic and simplification:
$= \text{transF}(m) \triangleright_O (\text{id} \triangleright_S \text{trans-spec}(\text{map-spec}))$
$= \text{transF}(m) \triangleright_S \text{trans-spec}(\text{map-spec})$
(9) By Lemma 9.4 and the definitions of linearizability and strict serializability:
$\lesssim_\mathbb{C} \text{atomic}(\text{trans-spec}(\text{interp-as-spec}(m \triangleright_S \text{map-spec})))$
(10) By Lemma 9.1:
$\lesssim_\mathbb{C} \text{atomic}(\text{trans-spec}(\text{interp-as-spec}(\text{pred-map} \triangleright_S (\text{ref-map-spec}_\phi + \text{map-spec}))))$
(11) By Lemma 9.2 and Lemma 4.1:
$\lesssim_\mathbb{C} \text{atomic}(\text{trans-spec}(\text{map-spec}))$
(12) By the definitions of linearizability and strict serializability:
$\lesssim_{\mathbb{SS}} \text{map-spec}$ □

10 COQ FRAMEWORK

We implemented the presented framework as a library in Coq called C4 (for Certified Composable Concurrency in Coq) [Lesani et al. 2022], that is made publicly available. As mentioned in Section 3, we represent programs as interaction trees [Xia et al. 2020], which are modeled in a coinductive type shown in Figure 18. Interaction trees are a mixed embedding [Chlipala 2021] that exposes exactly the behavior of objects that we care about, namely, the sequences of methods they call, while hiding irrelevant syntactic details: binders, conditionals, pattern-matching, and loops are handled by the metalanguage. Vertical composition and instrumentation are now functions on potentially infinite trees, which requires some care, but the upside is that they are agnostic to the constructs used to form loops. The interaction-tree library provides an equational theory (Lemma 4.6) that we leveraged in our verified transactional-predication application (Section 9).

```
CoInductive Prog M R :=
| Ret (_ : R)
| Tau (_ : Prog M R)
| Bind A (_ : M A) (_ : A → Prog M R).
```

Fig. 18. The type of interaction trees

Implementations Impl $M\ N$ are formalized as functions of type `∀ R, M R → itree N R`, mapping method calls in the high-level interface `M` to programs that perform method calls in the low-level interface `N`. A method call also determines a result type `R` that must be the result type of the program. (Implementations are called *handlers* in Xia et al. [2020].) Sequential specifications and objects are dependently typed records, respectively packaging a type of states with a transition relation and a low-level interface with a sequential specification and an implementation using it, as shown in Figure 19.

```
Definition Impl M N :=
  (∀ R, M R → Prog N R).
Record Spec M :=
  { State : Type
  ; Init : State
  ; Transitions : State → ∀R, M R →
       State → R → Prop }.
Record Object M :=
  { LowM : Type → Type
  ; ObjImpl : Impl M LowM
  ; ObjLowSpec : Spec LowM }.
```

Fig. 19. Implementations, sequential specifications, and objects in Coq

Linearizability, serializability, and associated constructions are defined as regular relations and functions in Coq, with signatures shown in Figures 6 and 13. The equations for composition operators (Section 4) and the proof principle for linearizability (Section 5) are the main tools that can be reused to reason about concurrent objects. To prove concurrent refinement (and linearizability), the user defines the invariants, instantiates the proof principle, and proves the proof obligations. Simplified instances of the proof principle without certain invariants that factor out boilerplate cases are also available.

Thanks to the executability of interaction trees, we can use Coq's extraction mechanism to translate implementations—the executable parts of objects—to Haskell and use them in multithreaded programs. We can interpret an interaction tree by a fold [Meijer et al. 1991] with an implementation of its method calls, using Haskell's concurrency primitives. However, explicitly manipulating tree structures adds significant run-time overhead. As future work, we would like to explore partial evaluation of extracted implementations to improve performance to a level competitive with concurrent data structures written directly in Haskell.

11 RELATED WORK

Composing concurrent operations. Several previous works present techniques to compose multiple calls on concurrent data structures into atomic operations. Transactional boosting [Herlihy and Koskinen 2008] benefits from commutativity specifications to allow concurrent execution of commutative calls and prevent concurrent execution of noncommuting calls by acquiring the same lock. Later works presented optimistic variants [Dickerson et al. 2019; Hassan et al. 2014].

Guerraoui [1995] introduced o-atomicity, a property of specifications of atomicity that allows multiple objects with different serialization orders to be composed in the same transaction, presenting sketches of both pessimistic and optimistic implementations that satisfy o-atomicity. Similarly, Reversible Atomic Objects (RAO) [Antonopoulos et al. 2016] propose an implementation technique for such compositions. In contrast to the two works above, this paper presents general and composable definitions to capture different specifications and implementation techniques modularly, plus proof principles to verify correctness. ROA can be captured in our framework as a composition instance, and the RAO refinement proof can be captured as linearizability of the resulting object with respect to the specification of the high-level interface. Finally, in contrast to RAO, our framework supports recursive method calls.

We saw transactional predication [Bronson et al. 2010] in this paper. Similarly, transactional data structures [Assa et al. 2020; Spiegelman et al. 2016] and transactional software objects [Herman et al. 2016] use TM judiciously and further benefit from specific data-structure semantics and organization to improve efficiency. Follow-up work presents lock-free variants [Elizarov et al. 2019]. In Foresight [Golan-Gueta et al. 2013], the client declares an overapproximation of the set of methods that they foresee to be called. To maintains a partial order for the composed operations, the library may temporarily block a method call that does not commute with the possible future calls by the environment. A few other works [LaBorde et al. 2019; Lamar et al. 2020; Zhang et al. 2018] present custom synchronization mechanisms to provide transactional implementations of specific data structures such as linked lists and vectors. However, the above do not provide proof techniques and formal atomicity guarantees.

Testing and verification of composed operations. Colt [Shacham et al. 2011] and ICFinder [Liu et al. 2013] test atomicity of, and Snowflake [Lesani et al. 2014] automatically verifies, composed methods that extend the interface of an already linearizable data structure [Lea 2000]. Our framework includes tactics that can automatically verify a strict superset of the above use cases. Further, it supports the definition of a more diverse set of objects including composition of multiple objects and transaction protocols, and it provides general proof techniques to verify them. Flint [Liu et al. 2014] fixes nonatomic composed methods. It infers a specification from the method itself and applies heuristics to synthesize a concurrent implementation. In contrast to Flint's repair of composed methods, our framework supports their formal definition and mechanized verification.

TxC-ADT [Peterson and Dechev 2017] generates happens-before graphs and applies model checking to check the consistency of transactional data structures. By contrast, our framework presents proof techniques to verify these data structures mechanically in a proof assistant.

Specification and verification of atomicity. Filipović et al. [2010] characterized linearizability as observational refinement. Attiya et al. [2017] characterized the TM correctness conditions TMS [Doherty et al. 2013] and opacity [Guerraoui and Kapalka 2008] as observational refinement. Thus, the notions of simulation and refinement [Abadi and Lamport 1991; Lynch and Vaandrager 1995] have been applied to verify atomicity [Bouajjani et al. 2017; Emmi and Enea 2019; Hawblitzel et al. 2015; Jagannathan et al. 2014; Kragl et al. 2020; Lesani et al. 2012a; Schellhorn et al. 2012; Turon et al. 2013]. However, they do not consider verification of composed operations. A related project [Armstrong et al. 2017] first proves the atomicity of individual operations (read, write, commit) of the transaction protocol and then applies simulation to prove serializability on top of that. In contrast to our formalism, it does not modularly state serializability in terms of linearizability and uses a standalone definition of opacity.

A related project [Cerone et al. 2014; Murawski and Tzevelekos 2019] presents multiple dedicated definitions of linearizability for objects (or libraries) that are implemented in terms of other objects. Our framework shows that a unique definition of linearizability, composition combinators, and proof technique can be the foundation of different instantiations including serializability. A few projects [Batty et al. 2013; Raad et al. 2019] consider modular verification of data structures on weak memory [Adve and Gharachorloo 1996; Sewell et al. 2010]. However, they do not consider transactions or the relation of linearizability and serializability. Further, the above projects did not consider the composition of transactional memory and concurrent data structures.

Modular systems. Objects and sequential specifications are similar to the modules and interfaces of certified abstraction layers [Gu et al. 2015], which were introduced in a sequential and deterministic context. Determinism enables proofs by downward simulation, and the sequential nature of modules allows them to support a flexible form of horizontal composition. In contrast, we leverage linearizable objects [Filipović et al. 2010; Gotsman and Yang 2012; Herlihy and Wing 1990] to build hierarchies of concurrent objects, our simulation proofs are upwards, and horizontal composition of concurrent objects requires their interfaces to be fully disjoint. In subsequent work on certified concurrent abstraction layers [Gu et al. 2018], interfaces specify behavior at the level of individual threads, whereas we focus on specifications using simple state machines that are agnostic to the number of threads interacting with objects.

Program logics. To our knowledge, while Hoare logics have long been applied in concurrent program verification, they have not been used for modular proof of examples combining classic concurrent data structures with transactional memory. However, many different extensions have been influential, including to our work. Rely-guarantee reasoning [Jones 1983] supports *temporal* decomposition of a workload across concurrent threads. The pioneering work on concurrent separation logic [Hobor et al. 2008; O'Hearn 2007] and its descendants [Windsor et al. 2017] tackled *spatial* decomposition of memory across separately verified data structures. Fruitful combination of these two techniques was demonstrated in the logics RGSep [Vafeiadis and Parkinson 2007], LRG [Feng 2009; Liang and Feng 2013] and TaDA [da Rocha Pinto et al. 2014]. This line of work relies on ghost state to formulate functional-correctness properties. A number of program logics rise to this challenge, by defining flexible higher-order ghost state connected to notions of state-transition systems. For instance, the logics FCSL [Nanevski et al. 2014] and Iris [Jung et al. 2016, 2015] build in different notions of monoids for expressing protocols [Liang and Feng 2013], and the GPS logic [Turon et al. 2014] applied similar ideas in the context of weak memory.

These logics support much more flexible state-sharing than in our framework. However, in return, we keep our framework much simpler and modularly build it from basic definitions, fixing little more than the classic notion of simulation and linearizability. Yet, we show that elaborate concurrent objects such as transaction protocols and predicated data structures can be implemented modularly. Further, in contrast to correctness conditions for transactions that were often written as standalone definitions [Doherty et al. 2013; Guerraoui and Kapalka 2008; Jagannathan et al. 2005; Papadimitriou 1979; Scott 2006], we show that serializability can be defined modularly based on linearizability and composition operations.

12 CONCLUSION

We have presented the first case study in formal verification that shows how to compose *verified transactional objects* whose implementations blend classic concurrent data structures with transactions. Moreover, while one might expect new complications in the reasoning framework to support this marriage, we demonstrated how it can be accomplished in a simple modular framework. Our notion of concurrent-object correctness is the classic one of *linearizability* with respect to sequential specifications, but applied in a higher-order logic with functional programs that manipulate higher-order structures. Our encoding of transactions relies on methods that take complete transactions as inputs and rewrite them syntactically to add synchronization, permitting us to state and prove strict serializability in terms of linearizability.

13 LIMITATIONS AND FUTURE WORK

The framework is first-order for objects and higher-order for programs. Programs (represented as interaction trees), but not objects, can be passed as arguments and returned. As a consequence, the interface of methods that an object may call is statically fixed.

Adding weak-memory primitives to the framework is future work. Although the focus of this paper was not on weak memory, the composition operators, their properties, and the statements of linearizability and of serializability in terms of linearizability are all general and independent of the implementation of the low-level objects at the bottom of the hierarchy. Many data structures that are implemented on weak memory provide linearizability at their interfaces and can be used and composed within this general framework.

Our framework models strict serializability in terms of linearizability. Opacity [Guerraoui and Kapalka 2008], another correctness condition for transactional memories, requires active in addition to completed transactions to observe consistent state. Previous work [Lesani et al. 2012b] defined a transition system (or specification) for opacity that constrains the return values of all calls from active transactions. Opacity can be modeled in our framework in terms of linearizability as well. A protocol object is opaque iff it is linearizable with respect to the opacity specification. A sketch of this idea is available in the appendix of the author's version.

Other transactional objects such as the TDSL transactional set [Spiegelman et al. 2016] similarly use an index map to accelerate traversals to locations that need to be updated. The index map provides the method `getPrev` that returns a node that can reach the predecessor of the node that needs to be updated. Although this method cannot be abstracted as a function, it can be abstracted as a simple relation between the key argument, the returned node, and the state of the list. We hope that similarly to our functional abstraction of the locator, this relational abstraction can help decompose verification of the index and the transactional list.

ACKNOWLEDGEMENTS

This work was funded by National Science Foundation grant 1942711 and grant CCF-1521584, for the Expedition on the Science of Deep Specification.

REFERENCES

Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theoretical Computer Science* 82, 2 (1991), 253–284.

Sarita V Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: A tutorial. *computer* 29, 12 (1996), 66–76.

Timos Antonopoulos, Paul Gazzillo, Eric Koskinen, and Zhong Shao. 2016. Vertical Composition of Reversible Atomic Objects. (2016).

Alasdair Armstrong, Brijesh Dongol, and Simon Doherty. 2017. Proving opacity via linearizability: a sound and complete method. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 50–66.

Gal Assa, Hagar Meir, Guy Golan-Gueta, Idit Keidar, and Alexander Spiegelman. 2020. Nesting and composition in transactional data structure libraries. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 405–406.

Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. 2017. Characterizing transactional memory consistency conditions using observational refinement. *Journal of the ACM (JACM)* 65, 1 (2017), 1–44.

Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library abstraction for C/C++ concurrency. *ACM SIGPLAN Notices* 48, 1 (2013), 235–248.

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. 2017. Proving linearizability using forward simulations. In *International Conference on Computer Aided Verification*. Springer, 542–563.

Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. Transactional predication: high-performance concurrent sets and maps for STM. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing* (Zurich, Switzerland) *(PODC '10)*. ACM, New York, NY, USA, 6–15. https://doi.org/10.1145/1835698.1835703

Andrea Cerone, Alexey Gotsman, and Hongseok Yang. 2014. Parameterised linearisability. In *International Colloquium on Automata, Languages, and Programming*. Springer, 98–109.

Adam Chlipala. 2021. Skipping the Binder Bureaucracy with Mixed Embeddings in a Semantics Course (Functional Pearl). *Proc. ACM Program. Lang.* 5, ICFP, Article 94 (aug 2021), 28 pages. https://doi.org/10.1145/3473599

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A logic for time and data abstraction. In *European Conference on Object-Oriented Programming*. Springer, 207–231.

Luke Dalessandro, Dave Dice, Michael Scott, Nir Shavit, and Michael Spear. 2010. Transactional mutex locks. In *European Conference on Parallel Processing*. Springer, 2–13.

Thomas Dickerson, Eric Koskinen, Paul Gazzillo, and Maurice Herlihy. 2019. Conflict Abstractions and Shadow Speculation for Optimistic Transactional Objects. In *Asian Symposium on Programming Languages and Systems*. Springer, 313–331.

Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. 2013. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing* 25, 5 (2013), 769–799.

Avner Elizarov, Guy Golan-Gueta, and Erez Petrank. 2019. LOFT: lock-free transactional data structures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 425–426.

Michael Emmi and Constantin Enea. 2019. Violat: generating tests of observational refinement for concurrent objects. In *International Conference on Computer Aided Verification*. Springer, 534–546.

Xinyu Feng. 2009. Local rely-guarantee reasoning. In *POPL '09*.

Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for concurrent objects. *Theoretical Computer Science* 411, 51-52 (2010), 4379–4398.

Guy Golan-Gueta, G. Ramalingam, Mooly Sagiv, and Eran Yahav. 2013. Concurrent Libraries with Foresight. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 263–274. https://doi.org/10.1145/2491956.2462172

Alexey Gotsman and Hongseok Yang. 2012. Linearizability with ownership transfer. In *International Conference on Concurrency Theory*. Springer, 256–271.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 595–608. https://doi.org/10.1145/2676726.2676975

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 646–661. https://doi.org/10.1145/3192366.3192381

Rachid Guerraoui. 1995. Modular atomic objects. *Theory and Practice of Object Systems* 1, 2 (1995), 89–99.

Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. 175–184.

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 48–60. https://doi.org/10.1145/1065944.1065952

Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. 2014. Optimistic transactional boosting. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 387–388.

Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. Automated and modular refinement reasoning for concurrent programs. In *International Conference on Computer Aided Verification*. Springer, 449–465.

Maurice Herlihy and Eric Koskinen. 2008. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (Salt Lake City, UT, USA) *(PPoPP '08)*. ACM, New York, NY, USA, 207–216. https://doi.org/10.1145/1345206.1345237

Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '93)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/165123.165164

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2016. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle semantics for concurrent separation logic. In *ESOP*.

Suresh Jagannathan, Vincent Laporte, Gustavo Petri, David Pichardie, and Jan Vitek. 2014. Atomicity refinement for verified compilation. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36, 2 (2014), 1–30.

Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony Hosking. 2005. A transactional object calculus. *Science of Computer Programming* 57, 2 (2005), 164–186.

Cliff B. Jones. 1983. Specification and Design of (Parallel) Programs. In *Information Processing 83*, Vol. 9. 321–332.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order Ghost State. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. ACM, New York, NY, USA, 256–269. https://doi.org/10.1145/2951913.2951943

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants As an Orthogonal Basis for Concurrent Reasoning. In *POPL '15* (Mumbai, India). 637–650.

Nicolas Koh, Yao Li, Yishuai Li, Li yao Xia, Lennart Beringer, Wolf Honore, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *8th ACM SIGPLAN International Conference on Certified Programs and Proofs*.

Bernhard Kragl, Shaz Qadeer, and Thomas A Henzinger. 2020. Refinement for structured concurrent programs. In *International Conference on Computer Aided Verification*. Springer, 275–298.

Pierre LaBorde, Lance Lebanoff, Christina Peterson, Deli Zhang, and Damian Dechev. 2019. Wait-Free Dynamic Transactions for Linked Data Structures. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores* (Washington, DC, USA) *(PMAM'19)*. Association for Computing Machinery, New York, NY, USA, 41–50. https://doi.org/10.1145/3303084.3309491

Kenneth Lamar, Christina Peterson, and Damian Dechev. 2020. Lock-free transactional vector. In *Proceedings of the Eleventh International Workshop on Programming Models and Applications for Multicores and Manycores*. 1–10.

Douglas Lea. 2000. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional.

Mohsen Lesani, Victor Luchangco, and Mark Moir. 2012a. A framework for formally verifying software transactional memory algorithms. In *International Conference on Concurrency Theory*. Springer, 516–530.

Mohsen Lesani, Victor Luchangco, and Mark Moir. 2012b. Putting opacity in its place. In *Workshop on the theory of transactional memory*. 137–151.

Mohsen Lesani, Todd Millstein, and Jens Palsberg. 2014. Automatic Atomicity Verification for Clients of Concurrent Data Structures. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag New York, Inc., New York, NY, USA, 550–567. https://doi.org/10.1007/978-3-319-08867-9_37

Mohsen Lesani, Li yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. 2022. *C4: Verified Transactional Objects*. https://doi.org/10.5281/zenodo.6342476

Hongjin Liang and Xinyu Feng. 2013. Modular Verification of Linearizability with Non-fixed Linearization Points. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 459–470. https://doi.org/10.1145/2491956.2462189

Peng Liu, Julian Dolby, and Charles Zhang. 2013. Finding Incorrect Compositions of Atomicity. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. ACM, New York, NY, USA, 158–168. https://doi.org/10.1145/2491411.2491435

Peng Liu, Omer Tripp, and Xiangyu Zhang. 2014. Flint: Fixing Linearizability Violations. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 543–560. https://doi.org/10.1145/2660193.2660217

N. Lynch and F. Vaandrager. 1995. Forward and Backward Simulations. *Information and Computation* 121, 2 (1995), 214 – 233. https://doi.org/10.1006/inco.1995.1134

Erik Meijer, Maarten Fokkinga, and Ross Paterson. 1991. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on functional programming languages and computer architecture*. Springer, 124–144.

Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L Hudson, Bratin Saha, and Adam Welc. 2008. Single global lock semantics in a weakly atomic STM. *ACM Sigplan Notices* 43, 5 (2008), 15–26.

Andrzej S. Murawski and Nikos Tzevelekos. 2019. Higher-order linearisability. *Journal of Logical and Algebraic Methods in Programming* 104 (2019), 86–116. https://doi.org/10.1016/j.jlamp.2019.01.002

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag New York, Inc., New York, NY, USA, 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307.

Christos H Papadimitriou. 1979. The serializability of concurrent database updates. *Journal of the ACM (JACM)* 26, 4 (1979), 631–653.

Christina Peterson and Damian Dechev. 2017. A Transactional Correctness Tool for Abstract Data Types. *ACM Trans. Archit. Code Optim.* 14, 4, Article 37 (Nov. 2017), 24 pages. https://doi.org/10.1145/3148964

Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. 2019. On library correctness under weak memory consistency: Specifying and verifying concurrent libraries under declarative consistency models. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–31.

Gerhard Schellhorn, Heike Wehrheim, and John Derrick. 2012. How to Prove Algorithms Linearisable. In *Proceedings of the 24th International Conference on Computer Aided Verification* (Berkeley, CA) *(CAV'12)*. Springer-Verlag, Berlin, Heidelberg, 243–259. https://doi.org/10.1007/978-3-642-31424-7_21

Michael Scott. 2006. Sequential specification of transactional memory semantics. (2006).

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97.

Ohad Shacham, Nathan Bronson, Alex Aiken, Mooly Sagiv, Martin Vechev, and Eran Yahav. 2011. Testing atomicity of composed concurrent operations. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (Portland, Oregon, USA) *(OOPSLA '11)*. ACM, New York, NY, USA, 51–64. https://doi.org/10.1145/2048066.2048073

Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (Ottowa, Ontario, Canada) *(PODC '95)*. Association for Computing Machinery, New York, NY, USA, 204–213. https://doi.org/10.1145/224964.224987

Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional Data Structure Libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, New York, NY, USA, 682–696. https://doi.org/10.1145/2908080.2908112

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 691–707. https://doi.org/10.1145/2660193.2660243

Aaron J Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 343–356.

Viktor Vafeiadis and Matthew Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*.

Matt Windsor, Mike Dodds, Ben Simner, and Matthew J. Parkinson. 2017. Starling: Lightweight Concurrency Verification with Views. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 544–569.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119 Distinguished paper award.

Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. 2020. An Equational Theory for Weak Bisimulation via Generalized Parameterized Coinduction. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*.

Deli Zhang, Pierre Laborde, Lance Lebanoff, and Damian Dechev. 2018. Lock-free transactional transformation for linked data structures. *ACM Transactions on Parallel Computing (TOPC)* 5, 1 (2018), 1–37.