Foundational and Compositional Verification of Layered Concurrent Objects

Yicheng Ni
Shanghai Jiao Tong University, School of Computer Science, China

Yuting Wang[1]
Shanghai Jiao Tong University, School of Computer Science, China

Abstract

Compositionality is essential for managing complexity in verification of concurrent programs, especially when proof certificates are needed. An effective verification scheme is to divide programs into abstraction layers and verify by composing refinements between layers. However, vanilla verified layers come with *global states* which severely limit their extensibility. This problem may be solved by dividing a layer into objects with *encapsulated local states*. Although the idea has been successfully realized in the sequential setting, it remains unclear if it also works in a concurrent setting where function calls across layers are no longer atomic steps and foundational proofs are needed.

We propose an approach to foundational and compositional verification of concurrent objects organized into multiple layers. The central idea is to represent concurrent objects as *open* labeled transition systems with *encapsulated threads* and to adopt *trace refinements* as a uniform notion of functional correctness which supports both *vertical composition* (layered refinement) and *horizontal composition* (extension of layer interfaces). Due to the operational nature of transition systems, it is straightforward to adopt traditional simulation techniques to produce foundational proof certificates. We implement this framework in the Rocq theorem prover and demonstrate its capability in foundational and compositional verification by verifying non-trivial lock-free concurrent data structures.

2012 ACM Subject Classification Theory of computation → Parallel computing models; Theory of computation → Program verification; Theory of computation → Operational semantics

Keywords and phrases Concurrency, Compositional Verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2026.21

Related Version *Extended Version*: https://doi.org/10.5281/zenodo.20640216

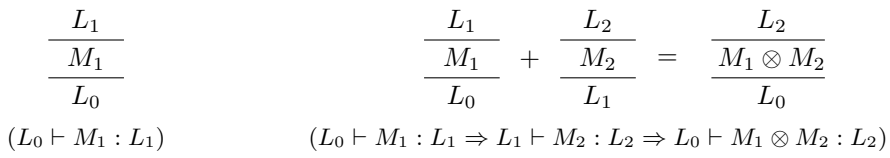
Supplementary Material *Software*: https://doi.org/10.5281/zenodo.19604812
Software (ECOOP 2026 Artifact Evaluation approved artifact):
https://doi.org/10.4230/DARTS.12.1.16

Funding This work was supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62372290, W2421088 and 62002217.

1 Introduction

Foundational verification aims at producing proof certificates for verified programs, with proofs mechanized in an interactive theorem prover (e.g., Rocq [57] or Lean [16]) and mechanically checked by a trusted kernel logic without relying on external automated tools such as SMT solvers, thereby promoting confidence in formal verification. Despite efforts devoted to verification techniques for concurrency, foundational verification of any non-trivial concurrent program remains notoriously difficult.

[1] Corresponding author



(a) A Single Layer. **(b)** Vertical Composition.

Figure 1 Verified Layers and Their Vertical Composition.

The main complexity in verifying non-trivial concurrent programs lies in reasoning about (non-)interference between concurrent operations on shared states. *Compositional reasoning* is key to manage this complexity, with which a concurrent program is divided into *basic units for reasoning* where each unit is verified independently, and the verification results are composed together to derive the correctness of the original program. The most prominent technique for compositional reasoning of concurrency is the axiomatic style Hoare logic proposed by Owicki and Gries [47] where *threads* are the basic units for reasoning. A Hoare triple is defined for each thread. By proving the non-interference between Hoare triples for threads, each thread can be independently verified like a *sequential* program. This technique is a foundational element of modern concurrent separation logics like VST [2] and Iris [29]. However, the flexibility to support *arbitrary program structures* in separation logics requires complicated verification techniques like frame rules [45, 51], ghost variables [28, 47] and rely-guarantee reasoning [27]. This severely limits its scalability to complex concurrent programs. As such, the state-of-the-art tools (e.g., VST and Iris and their various extensions) have not yet been testified against large programs like OS kernels [33].

1.1 Foundational and Compositional Verification of Layered Programs

One solution to the above problem is to avoid arbitrary program structures and instead focus on programs inherently coming with a compositional program structure, thereby exploiting simpler and more effective compositional verification techniques. In this work, we focus on foundational verification of programs that can be decomposed into *multiple abstraction layers*, such that a lower layer provides the *only* operations that its upper layer can perform. This kind of programs is ubiquitous in practice, especially for systems software where layered structures are the most natural abstraction for hiding implementation details.

Following this idea, formal verification has been successfully carried out at large scale for both sequential and concurrent systems software organized into layers. For the former, the most notable examples include Deep Specifications for building certified abstraction layers [21], FSCQ [7], a file system whose end-to-end crash safety is established by composing the proofs of individual layers, and Argosy [5], a compositional framework for verifying layered file systems via recovery refinement. For the latter, the most notable examples include CertiKOS, the first concurrent OS kernel with fine-grained locking that comes with a functional correctness proof [22], seKVM, a verified hypervisor [56], HyperEnclave whose page tables are verified using the layered approach [9], and the Civl verifier [34] which extends the Boogie verifier [3] to support verification of layers of concurrent programs. All that work comes with foundational proofs except for Civl.

Despite those successes, the layered approach alone is not compositional enough to support reuse and extension of existing verified programs, which we elaborate below. The basic unit in this approach is a *verified layer*, written as $L_0 \vdash M_1 : L_1$, denoting a module M_1 invoking APIs provided by a lower layer L_0 that in turn implements an upper layer L_1, as depicted

$$\frac{\dfrac{L_1}{M_1}}{L'} + \frac{\dfrac{L_2}{M_2}}{L'} = \frac{L_1 \otimes L_2}{M_1 \otimes M_2}{L'} \qquad \frac{\dfrac{L_1}{M_1}}{L'_1} + \frac{\dfrac{L_2}{M_2}}{L'_2} = \frac{L_1 \otimes L_2}{\dfrac{M_1 \otimes M_2}{L'_1 \otimes L'_2}}$$

(a) Existing Horizontal Composition. **(b)** Desired Horizontal Composition.

Figure 2 Horizontal Composition of Verified Layers.

in Figure 1a. Its correctness is usually defined as a refinement or simulation between the semantics of M_1 running on L_0 and that of L_1, denoted by $M_1 \otimes L_0 \sqsubseteq L_1$ where $\otimes$ is the linking operator. Modular verification is enabled by *vertical compositionality* of verified layers, as depicted in Figure 1b where $M_1 \otimes M_2$ denotes the composed (linked) module in which invocations between M_1 and M_2 through L_1 become internal calls. For example, the entire CertiKOS kernel is divided into 38 layers ($L_0, \ldots L_{37}$) such that $L_{i-1} \vdash M_i : L_i$ for $i \in \{1, \ldots, 37\}$ where L_0 provides the hardware interface and L_{37} exposes the OS APIs. By verifying each layer M_i and vertical compositionality, its correctness is established:

$$L_0 \vdash M_{CertiKOS} : L_{37} \text{ where } M_{CertiKOS} = M_1 \otimes \ldots \otimes M_{37}.$$

Now, suppose a third party would like to extend CertiKOS with a file system (which it currently lacks), which is formalized as $L_b \vdash M_f : L_f$ where L_b is the hardware interface M_f uses and L_f is the file system's APIs. If L_b is disjoint from L_0, one would expect to simply link the file system and CertiKOS side-by-side and get an extended kernel $L_0 \otimes L_b \vdash M_{CertiKOS} \otimes M_f : L_{37} \otimes L_f$. However, this is not possible because the naive layered approach assumes each layer contains a *global program state* (e.g., CertiKOS uses one global log for each layer [23]). If this global program state gets extended, then all the proofs about them need to be updated. With the above example, the third party would need to

1. Reprove CertiKOS with the extended lowest layer, i.e., prove $L_0 \otimes L_b \vdash M_{CertiKOS} : L_{37}$;
2. Reprove the file system with the extended lowest layer, i.e., prove $L_0 \otimes L_b \vdash M_f : L_f$;
3. Link them together to get $L_0 \otimes L_b \vdash M_{CertiKOS} \otimes M_f : L_{37} \otimes L_f$.

The first and second steps update and reprove the *invariants on the global state* with non-interference properties between L_b and *every layer* of CertiKOS. For linking to succeed, they should use the *same* invariants. All those involve an undesirable amount of work.

In summary, because of the existence of global states, verified layers could only support horizontal composition with the same lower layer (as depicted in Figure 2a), not with independent layers (as in Figure 2b). This limitation significantly undermines the value of verified layers, as they cannot be reused without breaching their modularity. This observation is general for naive verified layers and not limited to CertiKOS. For example, both the authors of Civl [34] and seKVM [40] observe that extension of verified layers requires substantial rewriting of proofs and relies heavily on automated proof generation to reduce manual efforts.

1.2 Challenges for Verifying Layers of Concurrent Objects

One solution to the above problem is to divide a layer with a single global state into objects encapsulating their local states underneath a set of operations so that *only* through them the local states can be accessed or modified. As a result, we get an even finer-grained compositional structure where non-interference between objects in a single layer may enable the desired horizontal composition. This idea has been successfully realized in the sequential setting. A notable example is the ConFrm framework [25] for verifying sequential file

Table 1 Frameworks for Foundational and Compositional Verification of Layered Programs.

Layer Components	SOTA Framework	Foundational	H. Comp	V. Comp	Concurrency
Global States	CertiKOS [22]	✓	✗	✓	✓
Sequential Objects	ConFrm [25]	✓	✓	✓	✗
Linearizable Objects	CompLin [46]	✗	✓	✓	✓
Concurrent Objects	**This Work**	✓	✓	✓	✓

systems that supports both vertical and horizontal compositions. ConFrm verifies layers by establishing *refinement mappings* [1], i.e., *deterministic* mappings from implementation states to specification states, by viewing function calls as atomic steps. Horizontal composition is achieved by composing independently verified objects operating on disjoint local states, and deriving a new refinement mapping from those for individual objects. Vertical composition is immediate because of the transitivity of refinement mappings which are essentially functions.

However, adapting this approach from sequential to concurrent settings faces significant challenges. In the presence of multiple threads and non-determinism, a concurrent program must be prepared for operations to be invoked and interleaved at arbitrary points during execution. Consequently, function calls can no longer be regarded as atomic steps. This necessitates mixing of simulation techniques beyond refinement mappings – such as forward, backward, or hybrid simulations with non-deterministic mappings to *sets* of states [26, 13] – for verifying layers with different levels of non-determinism. This not only complicates foundational verification where new concurrent semantic models and proof techniques are needed, but also makes it unclear if vertical and horizontal compositions still hold when a uniform simulation that is both vertically and horizontally compositional no longer exists.

A solution to supporting verified concurrent objects cannot be easily obtained by modifying existing layered verification frameworks for concurrency. In those frameworks, each layer is verified following the Owicki-Gries style [47] where threads are basic units of reasoning and rely-guarantee conditions of a single global state for threads are employed [27] (e.g., Civl uses *yield invariants* [36] to reason about thread interference on global states and CertiKOS [22] and subsequent verified hypervisors [56, 9] use global invariants to model interference between threads and CPUs). Obviously, this approach does not answer the question of how to prove non-interference between objects with isolated local states residing in a single layer.

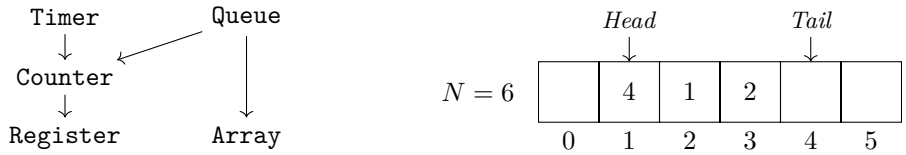
The most recent development for solving this problem is the compositional theory of linearizability [24] based on *game semantics* [46]. It focuses on linearizable concurrent objects and encodes them into inherently composable denotations of objects, i.e., *saturated strategies* in game models [18]. As such, vertical and horizontal compositions become possible through algebraic operations on strategies. However, this *denotational* approach makes foundational verification significantly more difficult. Previously, abstraction layers are given *operational* semantics, or even written as structured imperative programs (as in Civl [36]). Therefore, the well-known operational techniques (e.g., simulations) for proving refinements are directly applicable. With game semantics, specialized program logics are needed to prove denotational refinements between implementations and specifications [46], and complicated translations between strategies in the mathematical domain and program states are needed in *every step of reasoning* in the program logic. As such, the framework for compositional linearizability has not yet been formalized in a theorem prover.

1.3 Our Contributions

We propose an approach to addressing the challenges for verifying layers of concurrent objects, which leads to a framework that is both foundational and fully composable (both horizontally and vertically composable). As shown in Table 1, our framework overcomes the limitations of previous state-of-the-art (SOTA) approaches and supports all the desired features for the first time. Its key ideas are *1)* to treat concurrent objects as labeled transition systems (LTS) allowing for non-atomicity so that conventional operational reasoning is directly applicable to generated foundational proofs, and *2)* to obtain full compositionality by encapsulating both the state and control information of concurrent objects in LTS and using *trace refinements* [11] (i.e., inclusion of execution traces consisting of observable events generated by concurrent objects) of LTS as the uniform notion of correctness. The detailed contributions are described below (the full development is formalized in Rocq; see the supplementary materials).

- We introduce a novel form of labeled transition systems (LTS) that encapsulate both local states and control information (i.e., threads) of concurrent objects beneath interfaces that can both be invoked by upper layers and call into lower layers. It not only enables non-atomicity of concurrent function calls, but also enables hierarchical construction and horizontal composition of concurrent objects. As such, we could encode both concurrent objects (i.e., L in Figure 2) and its implementation (i.e., M in Figure 2) as LTS.
- Based on our semantic model, we develop a proof system for compositional verification of multiple layers of concurrent objects. The correctness of layered objects is formalized as trace refinements between LTS, i.e., inclusion of execution traces consisting of interface-level events generated by concurrent objects. We show that, by exploiting the operational structure of objects (instead of eliminating them as in denotational semantics), vertical and horizontal compositions of trace refinements are possible (because non-interference between threads and states in our LTS allows traces to be projected and combined along the LTS structures as we discuss in §6). That is, *full* compositionality for layered concurrent objects is achieved by using trace refinements as a uniform notion of correctness.
- We demonstrate that foundational verification of layered concurrent objects is possible by exploiting conventional operational techniques. In particular, trace refinements of LTS are derivable via either *forward* or *backward simulations*. In contrast to the denotational approaches, our framework keeps states and transitions explicit in the semantic model, enabling simulation-based proofs by avoiding complicated translation between traces and program states. In contrast to existing operational approaches (e.g., ConFrm), our framework does not rely on a particular form of simulation (which may only work in specific concurrent settings) and instead employs simple trace inclusions as the correctness definition, which works in any concurrent setting and enables full compositional reasoning.
- As case studies, we have verified a hierarchical implementation of concurrent timers based on forward simulation, and a lock-free bounded queue implemented using arrays based on backward simulation. Those studies show that our verified object layers are both reusable and extensible for modular construction of large concurrent objects from smaller ones.

In this paper, we consider concurrent programs with *preemptive* semantics in the setting of *sequential consistency*, i.e., atomic steps of concurrent processes may freely interleave with each other. In fact, our basic transition systems allow more liberal behaviors and obey sequential consistency only because of an intentionally placed constraint. It indicates that this work may be extended to work with weak memory model, which is left for future work. For generality of our approach, we avoid talking about specific programming languages by using transition relations over abstract states and pseudo code to describe programs.



(a) Dependency Graph. **(b)** A Lock-Free Bounded Queue.

Figure 3 A Running Example.

```
int reg; // value
bool CAS (old, new)
1. if reg == old then
       reg := new
2.     return true
3. return false

int READ ()
1. int v := reg
2. return v
```

(a) `Register`.

```
void INC ()
1. int a := READ()
2. int b := CAS(a,a+1)
3. if !b then goto 1
4. return

int GET ()
1. int w := READ()
2. return w
```

(b) `Counter`.

```
void TICK ()
1. INC()
2. return

int*int TIME ()
1. int t := GET()
2. int h := t / 60
3. int m := t mod 60
4. return (h, m)
```

(c) `Timer`.

Figure 4 Definitions of `Register`, `Counter` and `Timer`.

Connection with actual code requires further refinement and is also left for future work. Finally, we only investigate *safety* properties and leave liveness verification to future work.

In the rest of the paper, we first introduce the background and challenges in §2. We then present the key ideas in §3. In the subsequent three sections, we discuss our technical contributions. We then present evaluation in §7. Finally, we discuss related work and conclude in §8.

2 Background and Challenges

We first present an example that captures the essence of problems with global states and with verifying concurrent objects. We then discuss three challenges the existing approaches face, which exactly correspond to their limitations as described at rows 2–4 in Table 1.

2.1 A Running Example

The example concerns verification of two concurrent data structures in a layered fashion, as shown in Figure 3a. The first one is a clock (called `Timer`) with two operations: `TICK` increments time by one minute, and `TIME` returns the current time (in hours and minutes). To implement them, `Timer` calls an underlying `Counter` which also has two operations: `INC` and `GET` for incrementing and reading a natural number, respectively. `Counter` itself uses a `Register` (provided by hardware) to implement counting via compare-and-swap (`CAS`) instructions. The pseudo code for their implementations is shown in Figure 4. Note that, in this and remaining figures, we assume each atomic step (also called an *action*) is numbered (which may span multiple lines of code). Note also that the only shared state in Figure 4 is the global variable `reg` (the register value), with remaining variables local to each function.

The second data structure is a bounded queue (called `Queue`) which provides two op-

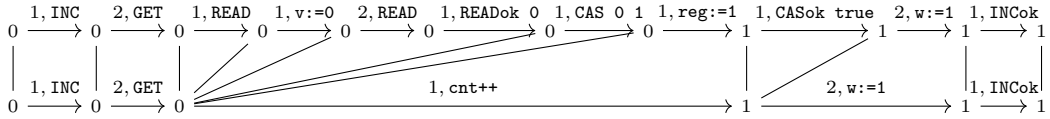


Figure 5 A Simulation between Implementation and Specification.

```
int cnt; // value
void INC ()
1. cnt++
2. return
```

```
int GET ()
1. int w := cnt
2. return w
```

Figure 6 Specification of `Counter`.

erations: `ENQ` for appending an element to the tail (if the queue is not full), and `DEQ` for removing and returning the head element (if the queue is not empty). `Queue` is given a lock-free implementation following [8]. It uses an array (called `Array` and given by hardware like registers) with a fixed size N and *two* `Counter` (modulo N) for recording head and tail indices of `Queue`, respectively. An example of bounded queue of size 6 is shown in Figure 3b, which is a snapshot after concurrent enqueue of $\{1, 2, 3, 4\}$ and dequeue of the first element (which happens to be 3). The code of `Queue` is given later in Figure 17 when it is needed.

On its own, `Timer` can be proven correct by layered refinement following the structure in Figure 3a, which in turn requires verifying `Counter`. However, the proof of `Counter` cannot be reused *as is* to prove `Queue`, for the same reason as we discuss the extension of CertiKOS with a file system in §1. To see that, let us first look at how `Timer` and `Counter` are verified.

2.2 Background: Layered Refinement and Vertical Composition

A concurrent program is *correct* if it *refines* its specification, meaning that all its possible behaviors are allowed by the specification. These behaviors are captured by the *preemptive* semantics of concurrent programs that describes interleaving of atomic execution steps (or *actions*) during execution. The first row of Figure 5 shows a sequence of interleaving execution steps for `Counter` when threads 1 and 2 invoke and execute the `INC` and `GET` operations. Each node represents the current counter value in `reg`. Each arrow is labeled with the thread identifier and the action performed. There are three kinds of actions: function calls (e.g., `INC`), function returns denoted with the suffix `ok` (e.g., `READok 0`), and internal steps (e.g., `v:=0`). Following this sequence, thread 1 completes the `INC` operation, updating `reg` to 1, while thread 2 reads the updated value but has not yet returned from `GET`.

To prove `Counter` correct, we define its specification in Figure 6. It abstracts `Register` by maintaining a global variable `cnt` for counting. `INC` and `GET` capture the intended behaviors: the former counts by 1, and the latter directly returns the counter value. An execution of threads 1 and 2 invoking `INC` and `GET` is shown in the second row of Figure 5. Each node contains the value of `cnt` and concurrent interleaving assumes atomic actions in Figure 6.

An implementation M refines its specification L, denoted by $M \sqsubseteq_R L$ if there exists a forward simulation with an invariant R relating states of M to L. For the counter example, we establish $M'_{cnt} \sqsubseteq_R L_{cnt}$ where M'_{cnt} denotes the combination of code in Figure 4a and 4b and L_{cnt} denotes Figure 6. Here, $R = \{(\texttt{reg}, \texttt{cnt}) \mid \texttt{reg} = \texttt{cnt}\}$, i.e., R is identity (also written as $R(\texttt{reg}) = \texttt{cnt}$). It is easy to check the execution of the implementation is simulated by the specification while preserving R, as Figure 5 shows where vertical lines represent R.

```
int  reg0, reg1;
bool CAS0 (old, new)
bool CAS1 (old, new)
int  READ0 ()
int  READ1 ()
```

(a) $L^0_{reg} \otimes L^1_{reg}$.

```
int cnt0, cnt1;
void INC0 ()
void INC1 ()
int  GET0 ()
int  GET1 ()
```

(b) $L^0_{cnt} \otimes L^1_{cnt}$.

$$\frac{\frac{L^0_{cnt}}{M^0_{cnt}}}{L^0_{reg}} + \frac{\frac{L^1_{cnt}}{M^1_{cnt}}}{L^1_{reg}} = \frac{\frac{L^0_{cnt} \otimes L^1_{cnt}}{M^0_{cnt} \otimes M^1_{cnt}}}{L^0_{reg} \otimes L^1_{reg}}$$

(c) Horizontal Composition.

Figure 7 Desired Horizontal Composition of `Counter`.

A verified layer, denoted by $L_0 \vdash_R M_1 : L_1$, is a concurrent program M_1 running on top of a specification L_0 to *correctly* implement the specification L_1, i.e.,

$$L_0 \vdash_R M_1 : L_1 \iff M_1 \otimes L_0 \sqsubseteq_R L_1.$$

We adopt the notation of Certified Concurrent Abstraction Layers (CCAL) [23]. Similar definitions are found in other works on layered refinement [34, 36, 35]. Then, $L_{reg} \vdash_R M_{cnt} : L_{cnt}$ holds where L_{reg} and M_{cnt} denote Figure 4a and Figure 4b, respectively.

With the implementation of counter abstracted into L_{cnt}, we can now prove the correctness of `Timer` by using L_{cnt}. Given a timer specification L_{timer}, we prove $L_{cnt} \vdash_S M_{timer} : L_{timer}$ where M_{timer} denotes Figure 4c and S is an invariant (L_{timer} is in our Rocq code). By vertical composition of the verified layers M_{cnt} and M_{timer}, we prove that $L_{reg} \vdash_{R \circ S} M_{timer} \otimes M_{cnt} : L_{timer}$ where $R \circ S$ is the transitive composition of R and S. This concludes that the complete implementation of timer in Figure 4 is correct with respect to its specification L_{timer}.

2.3 Challenge: Extending Verified Layers

However, existing approaches provide limited support for reusing or extending verified layers. Consider verifying the bounded queue in Figure 3, which makes use of two counters and an array. We naturally would like to exploit the already established correctness of `Counter`. That is, to compose two instances of `Counter` to prove the implementation of two counters (each relying on its own register) is correct with respect to their specification. Figure 7 illustrates this desired *horizontal composition* where a single counter (and register) layer is extended with another one by duplicating the state and method.

Unfortunately, such horizontal composition is not supported in verified layers with global states. One instead needs to update the global invariant from $R(\texttt{reg}) = \texttt{cnt}$ to $H(\texttt{reg}_0, \texttt{reg}_1) = (\texttt{cnt}_0, \texttt{cnt}_1)$ and go through the following proof steps as discussed in §1.1:

1. Prove $L^0_{reg} \otimes L^1_{reg} \vdash_H M^0_{cnt} : L^0_{cnt}$
2. Prove $L^0_{reg} \otimes L^1_{reg} \vdash_H M^1_{cnt} : L^1_{cnt}$
3. Compose them into $L^0_{reg} \otimes L^1_{reg} \vdash_H M^0_{cnt} \otimes M^1_{cnt} : M^0_{cnt} \otimes L^1_{cnt}$

To prove the first two steps, we need to show that there is no interference between M^0_{cnt} and L^1_{reg}, hence the execution of M^0_{cnt} maintains the invariant for L^1_{reg}. Although this may seem obvious in this example, it requires substantial changes to the old proof because function calls are not atomic in a concurrent setting. Furthermore, in more complicated situations, non-interference may be violated and results in failure of proofs. For example, this could happen if we replace $\texttt{GET}_i (i \in \{0, 1\})$ in Figure 7b with the following implementation:

```
int GETi ()
1. a = READi()
2. CAS1−i(42, 0) // modifies the other register
3. return a
```

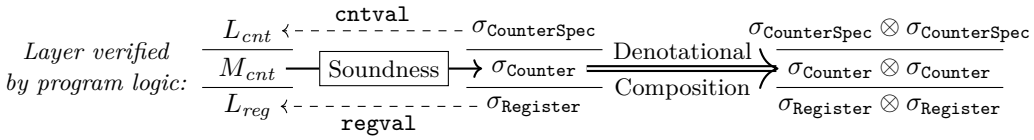


Figure 8 Verification of Two Counters in Compositional Linearizability.

2.4 Challenge: Verification of Encapsulated Objects with Concurrency

In our example, the state of a counter is *only* accessed by `INC` and `GET` (i.e., the ill-behaved case above does not happen). Therefore, non-interference between a counter and its environment is immediate if we could *encapsulate* the counter's state under `INC` and `GET`. That is, by dividing a verified layer into encapsulated and verified *objects*, we may achieve the desired horizontal composition in Figure 7c.

ConFrm [25] implements this solution for layered sequential programs. A verified object $L_0 \vdash M_1 : L_1$ generalizes a verified layer by encapsulating local states in L_i. Due to encapsulation, invariants for proving individual objects are hidden from the outside. Therefore, the desired horizontal composition is possible without reproving non-interference properties.

However, ConFrm is designed for sequential programs, in that its semantics treat function calls as atomic steps, which is not compatible with concurrent methods (e.g., those in Figure 4) that could be interrupted arbitrarily during execution. Therefore, refinements provided by ConFrm could not directly handle our examples. Furthermore, different simulation techniques are needed for proving concurrent objects with different levels of non-determinism. For example, `Counter` and `Timer` could be proved via a forward simulation with deterministic invariants (i.e., essentially a refinement mapping), while the lock-free bounded queue needs a much more complex backward simulation [8]. If different layers of objects make use of different simulation relations, we inevitably need to combine them into some uniform notion of simulation. However, there is at least a dozen of such simulations, none of which is agreed as a standard and all of which seem quite non-trivial (even in a sequential setting [6]).

2.5 Challenge: Foundational Verification of Concurrent Objects

Compositional theory of linearizability [46] provides one way to solve the above problem. It shows that *linearizability* could be characterized as a relation between *strategies* in game models. By interpreting concurrent objects as strategies (i.e., a set of traces they could produce), *locality* of linearizability [24] is proved. As locality denotes non-interference and state encapsulation for linearizable objects, the horizontal composition in Figure 7c becomes possible by treating all L and M as linearizable strategies, as depicted on the right of Figure 8.

Theoretically, this approach is capable of verifying layered concurrent objects. In reality, it is highly difficult to generate foundational proofs with this approach. Despite its denotational nature, the actual proofs are carried out operationally with a non-trivial embedding of a program logic based on operational semantics [31] into game semantics. The soundness of this logic ensures operational proofs indeed imply refinements in game semantics. To carry out operational reasoning in this logic, one must define *translation functions* which decode denotational traces to get back program states of objects. For example, the translation

```
class Register {
  int reg;
  CAS(old, new);
  READ(); }

bool CAS (old, new)
1. if reg == old then
       reg := new
2.     return true
3. return false

int READ ()
1. int v := reg
2. return v
```

(a) `Register`.

```
class Counter {
  Register rg;
  INC();
  GET(); }

void INC ()
1. int a := rg.READ()
2. int b := rg.CAS(a,a+1)
3. if !b then goto 1
4. return

int GET ()
1. int w := rg.READ()
2. return w
```

(b) `Counter`.

```
class Timer {
  Counter cnt;
  TICK();
  TIME(); }

void TICK ()
1. cnt.INC()
2. return

int*int TIME ()
1. int t := cnt.GET()
2. int h := t / 60
3. int m := t mod 60
4. return (h, m)
```

(c) `Timer`.

Figure 9 Definitions of `Register`, `Counter` and `Timer` Objects.

function for `Counter` in Figure 8 is defined as follows:

$$\texttt{cntval}(p) := \begin{cases} |\{i \mid \exists t,\ p_i = (t, \texttt{INC})\}|, & \forall i \forall t,\ p_i = (t, \texttt{INC}) \Rightarrow (\exists j > i,\ p_j = (t, \texttt{INCok}) \\ & \qquad \wedge\ (\nexists k,\ i < k < j \wedge p_k = (t, \texttt{INCok}))) \\ \bot, & \text{Otherwise} \end{cases}$$

where `cntval` translates any trace p that contains only complete `INC` and `INCok` pairs into the number of `INC` events in the trace as the value of `Counter` (where p_i denotes the ith event in p), and any other trace into a bottom value.

Using these auxiliary functions, the invariants are then formulated and proofs are built in the program logic. In fact, translation functions must be used in *every step* of proof building. This is already a lot of work for the small `Counter` example, and could get overwhelming for any non-trivial example (e.g., see page 71 of [46]). We conjecture that this complexity is why neither the compositional theory nor its accompanying case studies has yet been mechanized in a theorem prover at the time this paper is written.

3 Key Ideas

We discuss the key ideas for addressing the challenges in the previous section, so as to obtain an approach to verifying encapsulated objects that simultaneously support concurrency, foundational proofs and full compositionality. The overall idea is to combine the strengths of operational and denotational approaches, i.e., use operational semantics for encapsulating state and concurrency and use denotational style of refinement (i.e., trace refinement) which is proven both vertically and horizontally composable even in an operational setting.

3.1 Concurrent Transition Systems with Layered Interfaces

We represent concurrent objects as labeled transition systems (LTS) that encapsulate both object states and all concurrent accesses. Such an LTS has interfaces for interacting with upper and lower layers, thereby enabling vertical composition. Horizontal composition with other concurrent objects becomes possible thanks to state and concurrency encapsulation. Below we illustrate key features of LTS by using the example in Figure 4. We rewrite it into an object-oriented style in Figure 9 to emphasize state encapsulation, i.e., global variables in Figure 4 become member variables, and to make the dependency between objects clear.

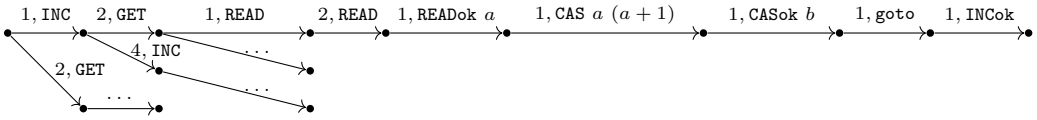
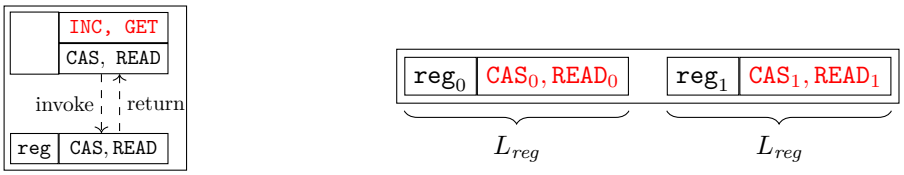


Figure 10 State Transitions of the `Counter` object.



(a) $L_{reg} \triangleleft M_{cnt}$. **(b)** $L_{reg} \otimes L_{reg}$.

Figure 11 Horizontal and Vertical Composition of Concurrent LTS.

Concurrent LTS with Layered Interfaces. An LTS $M : A \to B$ describes the behaviors of a concurrent object through small-step transitions where each step is the result of executing an atomic action (e.g., a numbered block in Figure 9). It is parameterized by two interfaces A and B where A describes *all* the possible operations M can perform on its lower layer and B describes the operations that M provides to its upper layer. For example, a `Counter` object in Figure 9b provides the `INC` and `GET` operations, and internally uses `CAS` and `READ` offered by an underlying `Register` object. It is represented as $M_{cnt} : I_{reg} \to I_{cnt}$, where $I_{reg} = (\{\texttt{CAS}\ o\ n, \texttt{READ}\}, \{\texttt{CASok}\ b, \texttt{READok}\ n\})$ and $I_{cnt} = (\{\texttt{INC}, \texttt{GET}\}, \{\texttt{INCok}, \texttt{GETok}\ n\})$. Each operation is divided into a *query* (e.g., `CAS o n`) and a *reply* (e.g., `CASok b`) where queries carry arguments and replies carry return values. Because we are modeling *preemptive* semantics, multiple threads can invoke the LTS arbitrarily, resulting in free interleaving of atomic actions. Figure 10 illustrates some of the behaviors of M_{cnt} starting from an initial state (the leftmost node) where each node denotes a state and each arrow (i, a) describes a small-step transition caused by thread i performing action a. Every path in Figure 10 denotes an interleaved execution trace. For example, the topmost path describes an interleaving of thread 1 executing `INC` to its completion and thread 2 partially executing `GET`. Note that only the instruction at line 3 in Figure 9b causes an internal transition; the remaining actions are all at the interface level. Because we assume sequential consistency, a thread runs in program order and may re-enter the object arbitrarily after the previous call returns. As we shall see, the explicit layer interfaces enable structured composition of transition systems.

Encapsulation of States and Threads. Our LTS maintains an explicit state which may be changed by internal transitions and hidden from the environment. That is, queries and replies in the interface (e.g., `CAS o n`, `CASok b`) do not mention internal states (e.g., register values). This enables composition of object states like ConFrm [25]. Our transition steps also explicitly record which threads are currently running inside an object. Only these threads could affect the internal state of an object. For example, in the top-most path in Figure 10, only threads 1 and 2 have accessed the counter. Combining with the fact that the environment can access an object only through its interface, we fully capture all the possible interference to the object's state *within* its LTS. As we shall see below, this enables both horizontal and vertical composition of verified concurrent objects.

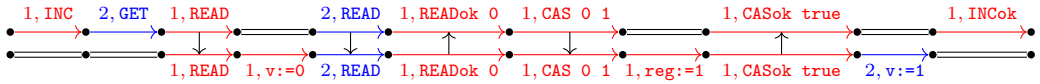


Figure 12 Vertical Composition of Traces for `Counter` and `Register`.

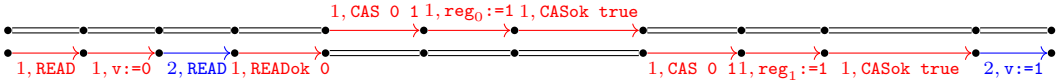


Figure 13 Horizontal Composition of Traces for Two `Registers`.

Vertical Compositionality. Two LTS $M_1 : A \to B$ and $M_2 : B \to C$ with a common layer interface B could be vertically composed into $M_1 \triangleleft M_2 : A \to C$, such that queries from M_2 into B are received by M_1 and replies from M_1 are received by M_2. For example, let $L_{reg} : \emptyset \to I_{reg}$ model the register object in Figure 9a. It is vertically composed with M_{cnt} to form $L_{reg} \triangleleft M_{cnt} : \emptyset \to I_{cnt}$. This is described in Figure 11a where an LTS consists of its internal state and interfaces such that only the methods in red are publicly visible. This composition hides the interaction through the register's interface, resulting in a complete implementation of counter that does not rely on any underlying layer and that exposes `INC` and `GET` operations. Figure 12 illustrates the composition of an execution trace of M_{cnt} with that of L_{reg} where events in different threads are marked with different colors. Note that queries and replies through the *middle layer* are *synchronized* where downward arrows denote calls and upward arrows denote replies. The remaining transitions remain unchanged, where double lines denote *stuttering steps* when one LTS is paused while the other is running.

Horizontal Compositionality. Two LTS $M_1 : A_1 \to B_1$ and $M_2 : A_2 \to B_2$ can be horizontally composed to form $M_1 \otimes M_2 : A_1 \otimes A_2 \to B_1 \otimes B_2$ where $\otimes$ denotes union of methods and states. For example, the horizontal composition of two L_{reg} is depicted in Figure 11b such that two internal registers and their accessing methods are paired together. The execution traces of the composed LTS are all the valid interleavings of traces of individual LTS. Figure 13 illustrates how two parallel execution traces of L_{reg} are composed. Here, thread 1 first performs `READ` on the second register, then performs `CAS` on the first register, and finally performs `CAS` for the second register. These steps are interleaved with the operations of thread 2. Note that, due to encapsulation provided by our LTS, separation and hiding of object states remain even after vertical or horizontal composition. This provides a foundation for compositional verification of concurrent objects as we shall discuss later.

Enforcing Sequential Consistency after Composition. Instruction reordering can happen if concurrent objects are horizontally or vertically composed without any restriction, hence violating sequential consistency. For example, let L_{cnt} denote the counter specification in Figure 15. Suppose $[(1, \texttt{INC}), (1, \texttt{INCok})]$ is a trace of L_{cnt} representing that thread 1 calls `INC` and returns. If we horizontally compose two L_{cnt} by naively interleaving traces, the composed object $L_{cnt} \otimes L_{cnt}$ may produce the trace $[(1, \texttt{INC}), (1, \texttt{INC}), (1, \texttt{INCok}), (1, \texttt{INCok})]$, indicating that thread 1 calls into the other L_{cnt} before returning from the first one. Such behavior violates the program order and hence sequential consistency. To solve this problem, we introduce an operator `sc` for instrumenting an LTS to explicitly record its active threads. The "real" horizontal composition $M_1 \mathbin{\overline{\otimes}} M_2$ operates on `sc` M_1 and `sc` M_2 and enforces

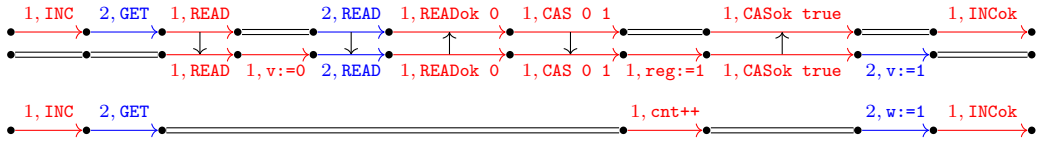


Figure 14 Trace Refinement between `Counter` and its Specification.

```
class Counter {
  int cnt;              void INC ()          int GET ()
  INC();                1. cnt++             1. int w := cnt
  GET(); }              2. return            2. return w
```

Figure 15 Specification of `Counter`.

the restriction that a thread may call one object only if it is not active in any other object. Similarly for vertical composition $M_1 \trianglelefteq M_2$. The updated composition operations rule out the invalid traces and ensure that composed LTS are always sequentially consistent.

3.2 Verified Concurrent Objects

Recall that the basic component of layered verification is a verified layer of the form $L \vdash_R M : L'$ indicating that a library layer L' is implemented by a program M invoking another library layer L. We generalize it to a *verified concurrent object* of the form $L \vdash M : L'$ in which a concurrent object L' is verified to be implemented by a concurrent program M invoking the underlying object L. Here, $L : \emptyset \to A$ and $L' : \emptyset \to B$ are *closed* objects, i.e., they do not depend on other objects (their underlying interface is empty), while $M : A \to B$ is an *open* object. We sometimes call the former an *object* and the latter an *implementation* (of an object) when it is obvious from context. Formally, a verified concurrent object is defined via trace refinement between the implementation and specification, as follows:

$$L \vdash M : L' \iff L \trianglelefteq M \sqsubseteq \mathtt{sc}\ L' \wedge L \sqsubseteq \mathtt{sc}\ L \wedge M \sqsubseteq \mathtt{sc}\ M.$$

A *trace refinement* $M \sqsubseteq M'$ holds if the *observable behaviors* of M, i.e., traces containing only events in the interfaces, are also the observable behaviors of M'. $L \sqsubseteq \mathtt{sc}\ L$ (or $M \sqsubseteq \mathtt{sc}\ M$) effectively means that L and $\mathtt{sc}\ L$ (or $M \sqsubseteq \mathtt{sc}\ M$) are observably equivalent since $\mathtt{sc}\ N \sqsubseteq N$ holds for any N. Therefore, L and M are both sequentially consistent. For example, we get the verified concurrent object $L_{reg} \vdash M_{cnt} : L_{cnt}$ by proving $L_{reg} \trianglelefteq M_{cnt} \sqsubseteq \mathtt{sc}\ L_{cnt}$. An example of this refinement is given in Figure 14 which shows the observable behavior of the composed implementation (i.e., $[(\mathtt{1,\ INC}), (\mathtt{2,\ GET}), (1, \mathtt{INCok})]$) also occurs in the specification. Verified concurrent objects enable foundational and compositional verification by exploiting the following features:

Compatibility with Operational Reasoning. . Verified concurrent objects are proved by directly employing operational techniques such as simulation. For example, to prove the trace refinement $L_{reg} \trianglelefteq M_{cnt} \sqsubseteq \mathtt{sc}\ L_{cnt}$, one simply uses the same invariant $R(\mathtt{reg}) = \mathtt{cnt}$ as before, and establish a *forward simulation* [43] between transitions in $L_{reg} \trianglelefteq M_{cnt}$ and L_{cnt}. For example, $(1, \mathtt{reg:=1})$ is simulated by $(1, \mathtt{cnt++})$ in Figure 14. More sophisticated simulation such as *backward simulation* [43] is needed for handling non-determinism in some concurrent programs, as we shall see for the bounded queue example. Nevertheless, because

$$\frac{L_1 \vdash M_1 : L_2 \qquad L_2 \vdash M_2 : L_3}{L_1 \vdash M_1 \vartriangleleft M_2 : L_3}\ \textsc{Vcomp} \qquad \frac{L_1 \vdash M_1 : L_1' \qquad L_2 \vdash M_2 : L_2'}{L_1 \mathbin{\overline{\otimes}} L_2 \vdash M_1 \mathbin{\overline{\otimes}} M_2 : L_1' \mathbin{\overline{\otimes}} L_2'}\ \textsc{Hcomp}$$

$$\frac{L_1 \vdash M_1 \vartriangleleft M_2 : L_2 \qquad M_1 \sqsubseteq \mathtt{sc}\ M_1}{L_1 \vartriangleleft M_1 \vdash M_2 : L_2}\ \textsc{Link} \qquad \frac{L_1 \vdash M_1 : L_2 \qquad L_1' \sqsubseteq \{L_1, \mathtt{sc}\ L_1'\}}{L_1' \vdash M_1 : L_2}\ \textsc{Weaken}$$

Figure 16 Composition Rules for Verified Concurrent Objects.

operational techniques are directly applicable, foundational verification becomes feasible without needing sophisticated embedding into denotational domains.

Encapsulation of Invariants. Verified concurrent objects enable more flexible and modular composition because they do not expose invariants. This is manifested by the fact that neither their definition nor trace refinements is indexed by an invariant. For example, $R(\mathtt{reg}) = \mathtt{cnt}$ is only used to establish simulation of internal steps; it is not needed to show the inclusion of observable events which do not mention internal states at all.

Trace Refinement as a Unified Correctness Condition. Because trace refinement only describes a relationship between observable events and could be derived from any kind of simulations, it serves as a unified correctness condition for verified concurrent objects. A key discovery of this work is that trace refinement is both vertically and horizontally composable if traces are generated from our concurrent LTS. This compositionality follows from non-interference between encapsulated threads and states, which in turn allows traces to be projected and combined along the structural composition of LTS. As a result, each concurrent object can be verified independently using the most suitable simulation and then composed together, as we illustrate below.

3.3 Composition of Verified Concurrent Objects

Compositional reasoning for verified concurrent objects is realized via the *sound* proof rules in Figure 16. Vcomp enables vertical composition of layers as described in Figure 1b except that a new vertical composition operator of LTS is used. Hcomp enables horizontal composition as described in Figure 2b which was not previously possible for vanilla verified layers. Notice that Hcomp makes use of $\overline{\otimes}$ instead of $\otimes$ to ensure that the horizontally composed objects are sequentially consistent. This is not necessary for Vcomp because *1)* by definition its premises ensure M_1 and M_2 are sequentially consistent, and *2)* the $\vartriangleleft$ operator preserves sequential consistency. The remaining rules are for deriving final refinement results for composed objects. In particular, the linking rule Link pushes the intermediate layer M_1 downwards and instantiates the underlying object that M_2 is built upon. The weakening rule Weaken enables replacement of an underlying layer with a more refined one where $L_1' \sqsubseteq \{L_1, \mathtt{sc}\ L_1'\}$ denotes $L_1' \sqsubseteq L_1$ and $L_1' \sqsubseteq \mathtt{sc}\ L_1'$.

A significant difference between our rules and existing work is that the former does not mention global state or global invariant. Therefore, they are more modular and succinct than before. For example, in CCAL (see Fig. 9 in [23]), horizontal composition employs rely-guarantee reasoning on global invariants; additional *compatibility* and *parallel composition* rules are needed for reasoning about multi-threading, which also employ global reasoning.

```
class Queue<N> {
  Counter front, rear;
  Array q<N>;
  ENQ(v);
  DEQ(); }
```

```
class Array<N> {
  elem ary[N];
  CAS(i, old, new);
  READ(i);
}
```

```
class Counter {
  Register rg;
  INC();
  GET();
}
```

```
class Register {
  int reg;
  CAS(old, new);
  READ();
}
```

```
void ENQ (v: val)
1. r := rear.GET()
2. x := q.READ(r mod N)
3. if r != rear.GET() then goto 1
4. if r == front.GET() + N then goto 1
5. if x.val != null then goto 1
6. if !q.CAS(r mod N,x,(v,x.ref+1))
      then goto 1
7. rear.INC()
8. return
```

```
val DEQ ()
1. f := front.GET()
2. x := q.READ(f mod N)
3. if f != front.GET() then goto 1
4. if f == rear.GET() then goto 1
5. if x.val == null then goto 1
6. if !q.CAS(f mod N,x,(null,x.ref+1))
      then goto 1
7. front.INC()
8. return x.val
```

Figure 17 Layered Implementation of a Lock-free Bounded Queue.

```
class QueueSpec<N> {
    list q;
    ENQ(v);
    DEQ();
}
```

```
void ENQ (v: val)
1. if #q == N
      then goto 1
      else q := q ++ [v]
2. return
```

```
val DEQ ()
1. if q == [h] ++ t
    then q := t
    else goto 1
2. return h
```

Figure 18 Specification of the Queue.

3.4 Verification of the Running Example

To get an intuition of how foundational and compositional verification work in our framework, we discuss the verification of our running example following the layered structure in Figure 3a.

We have already proved the correctness of `Counter` running on top of `Register` in §3.2, i.e., $L_{reg} \vdash M_{cnt} : L_{cnt}$. We finish verifying `Timer` by following the same process as described at the end of §2.2. That is, we define the specification of `Timer` as an object L_{timer} (its definition is omitted and can be found in our Rocq proofs). We then prove $L_{cnt} \vdash M_{timer} : L_{timer}$ by forward simulation where M_{timer} denotes Figure 9c. By applying VCOMP, we get $L_{reg} \vdash M_{cnt} \triangleleft M_{timer} : L_{timer}$, which is reduced to $L_{reg} \triangleleft M_{cnt} \vdash M_{timer} : L_{timer}$ by LINK. Finally, by unfolding its definition and stripping away `sc` by observing sequential consistency of objects, we get $L_{reg} \triangleleft M_{cnt} \triangleleft M_{timer} \sqsubseteq L_{timer}$. That is, the `Timer` program in Figure 9 correctly implements (refines) its specification L_{timer}.

We then verify the lock-free bounded queue (Figure 3b) whose code is presented in Figure 17. `Queue` is implemented by an `Array` object of size `N` which could be thought as a sequence of `Register` objects (i.e., with similar `CAS` and `READ` operations) implemented by hardware. A difference is that an element in the array is a structure of type `elem` with two fields: `val` for storing the actual value and `ref` for recording how many times this element has been modified. In particular, `val` could be `null` to indicate that no value is stored, and `ref` is a "version number" for distinguishing the same values appearing at different moments. `Queue` additionally employs two `Counter` objects as indices: `front` indexes the head of queue and `rear` indexes the tail of queue as shown in Figure 3b; both are used modulo `N`.

`Queue` provides an enqueue operation `ENQ` and a dequeue operation `DEQ`. `ENQ` pushes a value to its tail. At line 1 and 2, it gets the value `x` at the tail. At line 3 to 5, it checks if any concurrent `ENQ` has taken effect. If so, it restarts from line 1. At line 6, the method attempts to enqueue `v` by invoking `CAS` on the array. If it succeeds, it increments the tail index at line 7. `DEQ` follows a similar strategy to pop the head value.

Verification of `Queue` is carried out in the following steps. First, we prove that `Queue` object correctly implements a top-level specification L_{queue}, which formalizes code in Figure 18 where both `ENQ` and `DEQ` are atomic operations over a list of size `N`. By assuming specifications for its underlying counters and arrays, we need to prove

$$L_{cnt} \mathbin{\overline{\otimes}} L_{cnt} \mathbin{\overline{\otimes}} L_{ary} \vdash M_{queue} : L_{queue} \tag{1}$$

where L_{ary} denotes `Array` and M_{queue} denotes `Queue` in Figure 17. Because `Queue` contains more sophisticated non-determinism, this is proved by *backward simulation* with a set of highly non-trivial invariants on object states (see Example 20). Unlike the existing approaches, such complexity is never exposed to users thanks to encapsulation of invariants.

Second, by applying HCOMP on $L_{reg} \vdash M_{cnt} : L_{cnt}$ and $L_{ary} \vdash M_{id} : L_{ary}$ where M_{id} is an "copycat" implementation (i.e., identity), we get

$$L_{reg} \mathbin{\overline{\otimes}} L_{reg} \mathbin{\overline{\otimes}} L_{ary} \vdash M_{cnt} \mathbin{\overline{\otimes}} M_{cnt} \mathbin{\overline{\otimes}} M_{id} : L_{cnt} \mathbin{\overline{\otimes}} L_{cnt} \mathbin{\overline{\otimes}} L_{ary}. \tag{2}$$

Unlike the cumbersome rewriting shown in §2.3, we compose two verified counters as they are with minimum effort, thanks to encapsulation and isolation of interference between objects.

Finally, by vertical composition of (1) and (2) we get

$$L_{reg} \mathbin{\overline{\otimes}} L_{reg} \mathbin{\overline{\otimes}} L_{ary} \vdash (M_{cnt} \mathbin{\overline{\otimes}} M_{cnt} \mathbin{\overline{\otimes}} M_{id}) \triangleleft M_{queue} : L_{queue}. \tag{3}$$

By LINK and a sequence of WEAKEN, it is reduced to (where M_{id} is absorbed):

$$(L_{reg} \triangleleft M_{cnt}) \mathbin{\overline{\otimes}} (L_{reg} \triangleleft M_{cnt}) \mathbin{\overline{\otimes}} L_{ary} \vdash M_{queue} : L_{queue}. \tag{4}$$

By unfolding this definition and eliminating `sc`, we get the final trace refinement:

$$((L_{reg} \triangleleft M_{cnt}) \otimes (L_{reg} \triangleleft M_{cnt}) \otimes L_{ary}) \triangleleft M_{queue} \sqsubseteq L_{queue}. \tag{5}$$

That is, the code in Figure 17 correctly implements the atomic concurrent queue in Figure 18.

In summary, the above example shows that verified concurrent objects can be foundationally verified, reused as they are, and freely composed with other objects. In the subsequent three sections, we shall formally discuss the above key ideas in more depth.

4 Semantic Model

4.1 Formalization of Concurrent Labeled Transition Systems

We first formalize the labeled transition systems equipped with layered interfaces.

▶ **Definition 1** (Interface). *An interface is a tuple $A = (Q, R)$, where Q and R specify the set of query and reply actions, respectively.*

▶ **Definition 2** (Labeled transition system). *A labeled transition system $M : A \to B$ is a tuple $(S, N, I, \to, \to_i, \to_x, \to_y, \to_f)$ parameterized by interfaces A and B. S is the set of internal states; $N \subseteq S$ is the set of initial states; I is the set of internal actions. The following transition relations describe how threads in a set P execute within the system:*

- *$\to \subseteq S \times P \times I \times S$: internal steps, where $s \xrightarrow{p,j} s'$ means thread p performs an internal action $j \in I$ causing the internal state to transit from s to s';*
- *$\to_i \subseteq S \times P \times Q_B \times S$: calls from the higher-level interface (invocation transition); $s \xrightarrow{p,q_b}_i s'$ denotes that thread p enters from the upper layer by initiating a query $q_b \in Q_B$, causing the internal state to transit from s to s';*

$$\frac{i\ p = \texttt{None} \quad r\ p = \texttt{None}}{(i, r, v) \xrightarrow{p,(\texttt{CAS}\ o\ n)}_i ([p \mapsto (\texttt{CAS}\ o\ n)]i, r, v)}$$

(a) Invoke CAS.

$$\frac{i\ p = (\texttt{CAS}\ o\ n) \qquad o = v}{(i, r, v) \xrightarrow{p,\texttt{cas}} ([p \mapsto \texttt{None}]i, [p \mapsto (\texttt{CASok true})]r, n)}$$

(b) Execute CAS (successful case).

$$\frac{r\ p = (\texttt{CASok}\ b)}{(i, r, v) \xrightarrow{p,(\texttt{CASok}\ b)}_f (i, [p \mapsto \texttt{None}]r, v)}$$

(c) Return from CAS.

$$\frac{i\ p = (\texttt{CAS}\ o\ n) \qquad o \neq v}{(i, r, v) \xrightarrow{p,\texttt{cas}} ([p \mapsto \texttt{None}]i, [p \mapsto (\texttt{CASok false})]r, v)}$$

(d) Execute CAS (failed case).

Figure 19 Transition Rules for the `CAS` Operation.

- $\rightarrow_x \subseteq S \times P \times Q_A \times S$: *calls to the lower-level interface (external call transition);*
- $\rightarrow_y \subseteq S \times P \times R_A \times S$: *returns from lower-level calls (external return transition);*
- $\rightarrow_f \subseteq S \times P \times R_B \times S$: *returns to the higher-level interface (completion transition).*

The transition relations are organized according to the system's layered interfaces. They precisely characterize how a transition system interacts with higher-level systems (via $\rightarrow_i$ and $\rightarrow_f$) and with lower-level systems (via $\rightarrow_x$ and $\rightarrow_y$). A key difference between our LTS and the traditional ones (like in CertikOS and ConFrm) is that ours by definition captures only the threads running inside the LTS, i.e., thread behaviors are encapsulated with the LTS, which enables horizontal compositionality (see §6.2). Additionally, we prefer to use explicit transition relations (rather than a single transition relation with multiple labels) to model interactions with the environment because of the openness and layered nature of our LTS. Since an implementation may both invoke and be invoked by its environment, such transitions are separated into queries and replies. Furthermore, because an implementation may interact with both upper and lower layers, these transitions are further classified according to their layers. Thanks to such explicit structures, formal definitions of operations on our LTS become more structured and intuitive, as we shall see in the rest of §4.

▶ **Example 3** (Register object). `Register` in Figure 4a is encoded as L_{reg}. Its state (i, r, v) consists of the register value v and per-thread buffers i and r for pending invocations and responses. The transition rules of `CAS` in Figure 19 specify invocation and return by a thread p (first column) and the corresponding internal steps (second column).

▶ **Example 4** (Counter implementation). The `Counter` M_{cnt} additionally includes rules about external calls and returns, as illustrated in Figure 20, where the system's state is a tuple (pc, a, b, w) recording program counters and local variables for threads currently executing `INC` or `GET`. The first rule shows that when thread p reaches line 1 of `INC` (denoted by $\texttt{INC}_1$) in Figure 4b, it issues a `READ` query and waits for the reply ($\texttt{INC}'_1$ denotes the return address). The second rule shows p receives the reply and stores the return value n into its local variable.

$$\frac{pc\ p = \texttt{INC}_1}{(pc, a, b, w) \xrightarrow{p,\texttt{READ}}_x ([p \mapsto (\texttt{INC}'_1)]pc, a, b, w)}$$

(a) Call to External READ.

$$\frac{pc\ p = \texttt{INC}'_1}{(pc, a, b, w) \xrightarrow{p,(\texttt{READok}\ n)}_y ([p \mapsto (\texttt{INC}_2)]pc, [p \mapsto n]a, b, w)}$$

(b) Return from External READ.

Figure 20 Selected Transition Rules about External Call and Return in `INC`.

4.2 Mechanism for Ensuring Sequential Consistency

By definition, our LTS are quite liberal: they make no assumptions about the execution order of programs or threads. This could result in violation of sequential consistency as

$$\frac{s \xrightarrow{p,q_b}_{i'} s' \quad p \notin h}{(s,h) \xrightarrow{p,q_b}_{i} (s', [p \mapsto run]h)} \qquad \frac{s \xrightarrow{p,j}_{\prime} s' \quad h\ p = run}{(s,h) \xrightarrow{p,j} (s', h)} \qquad \frac{s \xrightarrow{p,q_a}_{x'} s' \quad h\ p = run}{(s,h) \xrightarrow{p,q_a}_{x} (s', [p \mapsto wait]h)}$$

$$\frac{s \xrightarrow{p,r_a}_{y'} s' \quad h\ p = wait}{(s,h) \xrightarrow{p,r_a}_{y} (s', [p \mapsto run]h)} \qquad \frac{s \xrightarrow{p,r_b}_{f'} s' \quad h\ p = run}{(s,h) \xrightarrow{p,r_b}_{f} (s', h/p)}$$

Figure 21 Transition Relations after Applying the `sc` Operator.

we discussed in §3.1. To solve this problem, we introduce a *sequential consistency operator* (`sc` −), which augments a transition system with an additional *thread state* h that records the states of threads currently executing within the system. Each such thread is either actively running (denoted *run*) or waiting (denoted *wait*) for the return of an external call, meaning that its control has been temporarily transferred to another transition system.

▶ **Definition 5** (Sequential consistency operator). *Given $M' : A \to B$, the* `sc` *operator turns $M' = (S', N', I', \to_{\prime}, \to_{i'}, \to_{f'}, \to_{x'}, \to_{y'})$ into a transition system $M =$* `sc` M' *:*

$$M := (S' \times H, N' \times \{h_0\}, I', \to, \to_i, \to_f, \to_x, \to_y),$$

where H is the set of functions of type $(P \to$ `option` $\{run, wait\})$ *and h_0 maps each thread to* `None`*, indicating that no thread is initially executing in the system. The transition relations in Figure 21 ensure sequential consistency by inspecting active threads in the thread state h.*

In the following sections, the sequential consistency operator will be used to define both horizontal and vertical composition, ensuring that the behaviors of threads in the composed system still satisfy sequential consistency.

4.3 Horizontal Composition

We define *horizontal composition* as combination of two transition systems into a new system that integrates the functionalities of both components. To preserve sequential consistency, we first apply `sc` to the components before combining them together.

▶ **Definition 6** (Horizontal composition). *Given transition systems $M_1' : A \to B$ and $M_2' : C \to D$ with $M_1 =$* `sc` M_1' *and $M_2 =$* `sc` M_2' *(where we assume $M_k = (S_k, N_k, I_k, \to_{\prime_k}, \to_{i_k}, \to_{f_k}, \to_{x_k}, \to_{y_k})$ for $k \in \{1, 2\}$), the horizontal composition of M_1' and M_2' is defined as:*

$$M_1' \mathbin{\overline{\otimes}} M_2' := (S_1 \times S_2, N_1 \times N_2, I_1 + I_2, \to, \to_i, \to_f, \to_x, \to_y),$$

with the extended interface $A \otimes C \to B \otimes D$, where $X \otimes Y = (Q_X + Q_Y, R_X + R_Y)$ for any two interfaces X and Y. The transition relations of horizontal composition are defined in Figure 22, where T_k denotes any transition relations excluding invocation transitions $\to_i$, t_k denotes actions that trigger T_k, and $s_k.h$ denotes the active threads in s_k.

By definition, the horizontally composed LTS accepts any initial call to its components and behaves like one of its components after that. The transition rules of horizontal composition ensure thread encapsulation: after composition each transition is contributed by exactly one thread in one component, while the other component's state remains unchanged. Horizontal composition also ensures that every thread is active only in a single LTS at any moment. In particular, we require that no thread may execute in both components simultaneously, ensured by the conditions in brackets ("[]") in the invocation rules in Figure 22.

$$\frac{s_1 \xrightarrow{p,q_b}_{i_1} s_1' \ [p \notin s_2.h]}{(s_1,s_2) \xrightarrow{p,q_b}_{i} (s_1',s_2)} \quad \frac{s_2 \xrightarrow{p,q_d}_{i_2} s_2' \ [p \notin s_1.h]}{(s_1,s_2) \xrightarrow{p,q_d}_{i} (s_1,s_2')} \quad \frac{s_1 \xrightarrow{p,t_1}_{T_1} s_1'}{(s_1,s_2) \xrightarrow{p,t_1}_{T} (s_1',s_2)} \quad \frac{s_2 \xrightarrow{p,t_2}_{T_2} s_2'}{(s_1,s_2) \xrightarrow{p,t_2}_{T} (s_1,s_2')}$$

Figure 22 Transition Rules for Horizontal Composition.

4.4 Vertical Composition

Vertical composition links objects in two adjacent layers by *synchronization*, which models interaction between LTS, and *hiding*, which abstracts away these interactions. It is similar to functional composition in game semantics [18], albeit applied to LTS instead of strategies.

We model synchronization as the control transfer of threads between two layers with a common interface. Consider two LTS $M_1 : A \to B$ and $M_2 : B \to C$ that share the interface B. When a thread p executing in M_2 triggers an external call, we model its control transfer to M_1 by triggering a simultaneous invocation transition in M_1. Conversely, when p reaches the end of a call into M_1 by performing a completion transition, its control transfers to M_2, represented by a simultaneous external return transition in M_2.

First, we formalize LTS with explicitly synchronized queries and replies called *synchronized labeled transition systems*, which serve as an intermediate form for vertical composition.

▶ **Definition 7** (Synchronized labeled transition system). *Given interfaces A, B and C, a synchronized LTS $G : A \to B \to C$ is a tuple $(S, N, I, \to, \to_i, \to_x, \to_y, \to_f, \to_q, \to_r)$.*

A synchronized LTS G exposes an interface with three layers: it offers interface C to the higher-level, relies on interface A from the lower-level, and interacts internally via interface B. The first eight components of G have the same meaning as in the standard LTS. Additionally, a synchronized LTS has two specialized relations: synchronized query transitions ($\to_q \subseteq S \times P \times Q_B \times S$) and synchronized reply transitions ($\to_r \subseteq S \times P \times R_B \times S$), which model interactions via interface B in the system.

▶ **Definition 8** (Synchronization). *Given $M_1' : A \to B$ and $M_2' : B \to C$ where $M_1 = \mathtt{sc}\ M_1'$ and $M_2 = \mathtt{sc}\ M_2'$ (we assume $M_k = (S_k, N_k, I_k, \to_k, \to_{i_k}, \to_{f_k}, \to_{x_k}, \to_{y_k})$ for $k \in \{1,2\}$), the synchronization of M_1' and M_2' results in a synchronized LTS $M_1' \overline{\|} M_2' : A \to B \to C$:*

$$M_1' \overline{\|} M_2' := (S_1 \times S_2, N_1 \times N_2, I_1 + I_2, \to, \to_i, \to_x, \to_y, \to_f, \to_q, \to_r),$$

where the transition rules are defined in Figure 23 (internal transitions are omitted and performed by running either M_1 or M_2):

- *the first column specifies invocation and completion transitions controlled by M_2;*
- *the second column specifies external call and return transitions controlled by M_1;*
- *the third column specifies the synchronized query transition triggered by the simultaneous external call of M_2 and invocation of M_1, and the synchronized reply transition triggered by the simultaneous completion of M_1 and external return of M_2.*

▶ **Example 9** (Synchronized counter). The `Counter` implementation M_{cnt} can be synchronized with the `Register` object L_{reg} to form $L_{reg} \overline{\|} M_{cnt}$. For example, the transition in Figure 12 is derived by synchronizing the shared actions (e.g., `READ` and `READok`).

As indicated by their multilayer interfaces, the synchronized query and reply transitions are still observable to the environment in synchronized transition systems, which provides information for deriving vertical compositionality (see §6.3). However, these interactions are desired to be abstracted away when this composite system is used as a building block for more complex systems. We formalize this abstraction as hiding.

$$\frac{s_2 \xrightarrow{p,q_c}_{i_2} s_2' \quad s_2 \xrightarrow{p,r_c}_{f_2} s_2'}{(s_1,s_2) \xrightarrow{p,q_c}_{i} (s_1,s_2') \quad (s_1,s_2) \xrightarrow{p,r_c}_{f} (s_1,s_2')} \qquad \frac{s_1 \xrightarrow{p,q_a}_{x_1} s_1' \quad s_1 \xrightarrow{p,r_a}_{y_1} s_1'}{(s_1,s_2) \xrightarrow{p,q_a}_{x} (s_1',s_2) \quad (s_1,s_2) \xrightarrow{p,r_a}_{y} (s_1',s_2)}$$

$$\frac{s_2 \xrightarrow{p,q_b}_{x_2} s_2' \quad s_1 \xrightarrow{p,q_b}_{i_1} s_1' \quad s_1 \xrightarrow{p,r_b}_{f_1} s_1' \quad s_2 \xrightarrow{p,r_b}_{y_2} s_2'}{(s_1,s_2) \xrightarrow{p,q_b}_{q} (s_1',s_2') \quad (s_1,s_2) \xrightarrow{p,r_b}_{r} (s_1',s_2')}$$

Figure 23 Transition Rules for Synchronization.

$$\frac{s \xrightarrow{p,j}_{\prime} s'}{s \xrightarrow{p,j} s'} \qquad \frac{s \xrightarrow{p,q_b}_{q'} s'}{s \xrightarrow{p,q_b} s'} \qquad \frac{s \xrightarrow{p,r_b}_{r'} s'}{s \xrightarrow{p,r_b} s'}$$

Figure 24 Internal Transition of Hiding.

$$\frac{}{s \xrightarrow{[]}_{*} s} \qquad \frac{s \xrightarrow{es}_{*} s'' \quad s'' \xrightarrow{p,j} s'}{s \xrightarrow{es}_{*} s'} \qquad \frac{s \xrightarrow{es}_{*} s'' \quad s'' \xrightarrow{p,w}_{T'} s'}{s \xrightarrow{es;(p,w)}_{*} s'}$$

Figure 25 Execution of Transition Systems.

▶ **Definition 10** (Hiding). *Given a synchronized LTS* $G' : A \to B \to C = (S', N', I', \to_{\prime}, \to_{i'}, \to_{f'}, \to_{x'}, \to_{y'}, \to_{q'}, \to_{r'})$*, the hiding* $(\circ -)$ *of* G' *is a standard LTS:*

$$\circ G' : A \to C := (S', N', I' + Q'_B + R'_B, \to, \to_{i'}, \to_{f'}, \to_{x'}, \to_{y'}),$$

with $\to$ *defined in Figure 24, reducing synchronized query and reply into internal transitions.*

Finally, we define vertical composition as the combination of synchronization and hiding.

▶ **Definition 11** (Vertical composition). *Given* $M_1 : A \to B$ *and* $M_2 : B \to C$*, the vertical composition* $(- \triangleleft -)$ *of* M_1 *and* M_2 *is the LTS* $M_1 \triangleleft M_2 := \circ(M_1 \overline{\|} M_2)$*.*

5 Correctness of Verified Concurrent Objects

We now formally define *verified concurrent objects* using trace refinement between concurrent LTS. We then discuss how to prove them correct using simulation techniques.

As our transition systems interact with the environment only via interfaces, their traces (i.e., sequences of queries and replies) capture all observable behaviors without exposing internal states. Therefore, we are able to adopt standard *trace refinement* as the correctness condition, i.e., inclusion of traces produced by LTS.

▶ **Definition 12** (Event). *Given an interface* A*, an event is a pair* $e = (p, w)$ *where* $p \in P$ *is a thread and* $w \in Q_A + R_A$ *is a query or reply.*

▶ **Definition 13** (Execution). *Given a (synchronized) LTS* M*, an execution* $s \xrightarrow{es}_{*} s'$ *in* M*, defined inductively as in Figure 25, represents a sequence of transitions from* s *to* s' *that generates events* es*. Here,* $\to_{T'}$ *denotes any non-internal transition, i.e.,* $\to_{i/f/x/y/q/r}$*.*

▶ **Definition 14** (Trace). *Given a (synchronized) LTS* M *whose initial state space is* N*, a trace* t *is an event sequence generated by an execution* $s \xrightarrow{t}_{*} s'$*, where* $s \in N$*.*

▶ **Definition 15** (Trace refinement). *Given* $M_1, M_2 : A \to B$*, we say* M_1 trace refines M_2*, denoted by* $M_1 \sqsubseteq M_2$*, if for any trace* t *of* M_1*,* t *is also a trace of* M_2*.*

A direct application of refinement is to formally specify properties that a transition system should satisfy, such as the *sequential consistency* of transition systems as follows.

▶ **Definition 16** (Sequential consistency). *An LTS* M *is* sequentially consistent *if* $M \sqsubseteq (\mathtt{sc}\ M)$*.*

We define verified concurrent objects, requiring that the vertical composition of an implementation M with a lower-level object L trace refines its specification L'.

▶ **Definition 17** (Verified concurrent object). *Given $L : \phi \to A$, $M : A \to B$, and $L' : \phi \to B$, we write $L \vdash M : L'$ if (1) L and M are sequentially consistent, and (2) $L \triangleleft M \sqsubseteq \mathtt{sc}\, L'$.*

Verified concurrent objects can be established using standard operational techniques such as simulations. Moreover, generality of trace refinement allows different simulation techniques to be applied in different situations. Below we formalize *backward simulation*, adapted from standard automata theories [43] to illustrate this point (See the extended version for the definition of forward simulation and the detailed proof of backward simulation).

▶ **Definition 18** (Backward Simulation). *For $L : \phi \to A$, $M : A \to B$ and $L' : \phi \to B$, the backward simulation between $L\overline{\|}M$ and L', written as $(L\overline{\|}M) \sim_b L'$, is a relation b over S_1 (states of $L\overline{\|}M$) and S_2 (states of L') s.t. ($\to_{i/f}$ denotes $\to_i$ or $\to_f$, and similarly for $\to_{q/r}$):*

1. *For any state s_1, there exists s_2 such that $s_1\ b\ s_2$,*
2. *If $s_1 \in N_1$ and $s_1\ b\ s_2$, then $s_2 \in N_2$,*
3. *If $s_1' \xrightarrow{p,w}_{i/f} s_1$ and $s_1\ b\ s_2$, then there exists s_2' s.t. $s_2' \xrightarrow{p,w}_{i/f} s_2$ with $s_1'\ b\ s_2'$,*
4. *If $(s_1' \xrightarrow{p,j} s_1$ or $s_1' \xrightarrow{p,w}_{q/r} s_1)$ and $s_1\ b\ s_2$, then there exists s_2' s.t. $s_2' \xrightarrow{\tau}_* s_2$ with $s_1'\ b\ s_2'$.*

In general, backward simulation can be defined between any two LTS. We present a specialized version for synchronized systems, since in practice one LTS is the vertical composition of an implementation with its underlying object (see Theorem 19). In this case, we often use invariants of synchronized systems, stating that in a synchronized query transition, the implementation and the object share the same arguments for a function call.

From backward simulation we derive verified concurrent object as follows.

▶ **Theorem 19** (Backward Simulation). *Given sequentially consistent systems $L : \phi \to A$, $M : A \to B$ and $L' : \phi \to B$, if there exists a relation b s.t. $(L\overline{\|}M) \sim_b L'$, then $L \vdash M : L'$.*

Intuitively, to prove $L \vdash M : L'$, we need to show $L \triangleleft M \sqsubseteq L'$, which can be divided into two steps following the definition of $L \triangleleft M$. First, $(L\overline{\|}M) \sim_b L'$ ensures that every event in the interface B generated by $(L\overline{\|}M)$ can also be generated by L'. Second, $\circ(L\overline{\|}M)$ hides events in A generated by $(L\overline{\|}M)$. Note that, the simulation invariant b is not exposed in verified concurrent objects, which enables their flexible compositionality as we shall see in §6.

▶ **Example 20.** To establish $L_{cnt} \mathbin{\overline{\otimes}} L_{cnt} \mathbin{\overline{\otimes}} L_{ary} \vdash M_{queue} : L_{queue}$ in §3.4, we can prove the backward simulation between $(L_{cnt} \mathbin{\overline{\otimes}} L_{cnt} \mathbin{\overline{\otimes}} L_{ary})\overline{\|}M_{queue}$ and L_{queue} with the invariant $U = \{((\mathtt{cnt}_0, \mathtt{cnt}_1, \mathtt{ary}, h, t), \mathtt{q}) \mid \mathit{gather}(h \bmod \mathtt{N}, t \bmod \mathtt{N}, \mathtt{ary}) = \mathtt{q}\}$, where

$$\mathit{gather}(a, b, \mathtt{ary}) := \begin{cases} \mathit{ary}[a \ldots (b-1)], & \text{if } a \le b, \\ \mathit{ary}[a \ldots (N-1)] \mathbin{@} \mathit{ary}[0 \ldots (b-1)], & \text{otherwise.} \end{cases}$$

Here, h and t are auxiliary state variables of M_{queue} recording the actual head and tail indices. They are needed because $\mathtt{cnt}_0$ and $\mathtt{cnt}_1$ are incremented (line 7 in Figure 17) after updating `ary` (line 6), so the queue contents cannot be directly reconstructed from these counters. The *gather* function collects elements of `ary` from a to $(b-1)$ (inclusively) if $a \le b$, and otherwise collects from a to the end of the array and then from index 0 to $(b-1)$.

To prove backward simulation, we must show that U is preserved by each step of $(L_{cnt} \mathbin{\overline{\otimes}} L_{cnt} \mathbin{\overline{\otimes}} L_{ary}\overline{\|}M_{queue})$. The intricate case arises when `Array` performs a successful `CAS` (line 6) where an invariant is needed stating that a `CAS` implies that the local variable r in `ENQ`, or f in `DEQ`, equals t or h, respectively. In other words, r or f holds the latest index, indicating that no concurrent update has occurred. This invariant mutually depends on about a dozen of other invariants that together capture the key properties of the queue.

6 Horizontal and Vertical Composition of Verified Concurrent Objects

We show that verified concurrent objects support horizontal and vertical composition enabled by encapsulation of our transition systems. Before that, we first introduce *plain* composition operators (§6.1), the basis for establishing compositionality.

6.1 Decoupling Sequential Consistency from Composition Operators

The composition operators ($- \mathbin{\overline{\otimes}} -$ and $- \mathbin{\overline{\triangleleft}} -$) apply $\mathtt{sc}$ to enforce sequential consistency. This raises two issues. First, traces of M that violate sequential consistency are silently removed by $\mathtt{sc}\, M$, so verifying $\mathtt{sc}\, M$ does not convincingly establish properties of M itself. Second, each composition reapplies $\mathtt{sc}$, introducing unnecessary reasoning overhead.

To address the first issue, we treat sequential consistency as a property to be verified, rather than an inherent assumption about every transition system. Consequently, we decouple the $\mathtt{sc}$ operator from the composition operators, which also solves the second issue. Below, we define such *plain* composition operators and establish refinements with the original ones.

▶ **Definition 21** (Plain horizontal composition). *Given $M_1 : A \to B$ and $M_2 : C \to D$, their plain horizontal composition is $M_1 \otimes M_2 = (S_1 \times S_2, N_1 \times N_2, I_1 + I_2, \to, \to_i, \to_f, \to_x, \to_y)$ where the transition relations are defined in Figure 22 with the conditions in brackets ignored.*

We call this *plain* horizontal composition as it does not concern itself with consistency of threads during execution. Similarly, we define plain synchronization and vertical composition.

▶ **Definition 22** (Plain synchronization). *Given $M_1 : A \to B$ and $M_2 : B \to C$, $M_1 \| M_2 = (S_1 \times S_2, N_1 \times N_2, I_1 + I_2, \to, \to_i, \to_f, \to_x, \to_y, \to_q, \to_r)$ with transition rules defined in Figure 23.*

▶ **Definition 23** (Plain vertical composition). *Given M_1 and M_2, $M_1 \lhd M_2 := \circ(M_1 \| M_2)$.*

We decouple the $\mathtt{sc}$ operator from composition by the following extraction lemmas (See the extended version of the paper for the detailed proofs of extraction lemmas).

▶ **Proposition 24** (Horizontal extraction). *If $M_1 : A \to B$ and $M_2 : C \to D$ are sequentially consistent, then $\mathtt{sc}\,(M_1 \otimes M_2) \sqsubseteq M_1 \mathbin{\overline{\otimes}} M_2$.*

▶ **Proposition 25** (Vertical extraction). *If $M_1 : A \to B$ and $M_2 : B \to C$ are sequentially consistent, then $\mathtt{sc}\,(M_1 \lhd M_2) \sqsubseteq M_1 \mathbin{\overline{\lhd}} M_2$.*

6.2 Horizontal Compositionality of Verified Concurrent Objects

We establish horizontal composition by proving soundness of HCOMP in Figure 16 (soundness of VCOMP is in §6.3, and the remaining rules are straightforwardly sound). As stated in Theorem 26, two verified concurrent objects relying on separate lower-level objects can be horizontally composed into a new verified concurrent object. This follows from encapsulation of threads and invariants in our transition systems, which prevents interference across objects.

▶ **Theorem 26** (Horizontal Compositionality). *If $L_1 \vdash M_1 : L_1'$ and $L_2 \vdash M_2 : L_2'$, then $L_1 \mathbin{\overline{\otimes}} L_2 \vdash M_1 \mathbin{\overline{\otimes}} M_2 : L_1' \mathbin{\overline{\otimes}} L_2'$.*

By unfolding the definitions, Theorem 26 is reduced to proving that trace refinement is preserved after horizontal composition, as shown in Lemma 27.

▶ **Lemma 27** (Horizontal preservation). *Given transition systems* $L_1, L_1' : \phi \to A$ *and* $L_2, L_2' : \phi \to B$*, if* $\mathtt{sc}\, L_1 \sqsubseteq \mathtt{sc}\, L_1'$ *and* $\mathtt{sc}\, L_2 \sqsubseteq \mathtt{sc}\, L_2'$*, then* $L_1 \mathbin{\overline{\otimes}} L_2 \sqsubseteq L_1' \mathbin{\overline{\otimes}} L_2'$.

This can be derived by further proving that the execution of a horizontally composed system can be *projected* to the execution of its two components, and that the execution of components can be again *combined* to form an execution of the composite system, as follows.

Horizontal projection states that an execution of the horizontal composition can be decomposed into the executions of its two components ($\circ_X es$ denotes the subsequence of es consisting only of events in X):

▶ **Lemma 28** (Horizontal projection). *For* $L_1 : \phi \to A$*,* $L_2 : \phi \to B$*, and* es*, if* $(s_1, s_2) \xrightarrow{es}_* (s_1', s_2')$ *is an execution in* $L_1 \mathbin{\overline{\otimes}} L_2$*, then* $s_1 \xrightarrow{\circ_A es}_* s_1'$ *in* $\mathtt{sc}\, L_1$ *and* $s_2 \xrightarrow{\circ_B es}_* s_2'$ *in* $\mathtt{sc}\, L_2$.

Horizontal combination states that the executions of two transition systems can be interleaved to form an execution of their horizontal composition. It is more subtle than horizontal projection because we must ensure that every event in the interleaved trace is admissible in the horizontally composed system, which preserves sequential consistency. We introduce the notion of *horizontal consistency* ($h_1 | h_2 \vdash_h es$) to capture such constraints (See the extended version for the definition of horizontal consistency).

▶ **Lemma 29** (Horizontal combination). *For* $L_1 : \phi \to A$*,* $L_2 : \phi \to B$*, and* es*, if* $s_1 \xrightarrow{\circ_A es}_* s_1'$ *in* $\mathtt{sc}\, L_1$*,* $s_2 \xrightarrow{\circ_B es}_* s_2'$ *in* $\mathtt{sc}\, L_2$*, and* $s_1.h | s_2.h \vdash_h es$*, then* $(s_1, s_2) \xrightarrow{es}_* (s_1', s_2')$ *in* $L_1 \mathbin{\overline{\otimes}} L_2$.

Intuitively, the key of the two lemmas is that each transition of the composed system originates from exactly one component, and the components do not interfere with each other. The same non-interference property holds for vertical composition, as we shall see next.

6.3 Vertical Compositionality of Verified Concurrent Objects

The vertical composition rule VCOMP in Figure 16 is formalized as Theorem 30, i.e., if one verified concurrent object implements the lower-level specification of the other, they can be composed by hiding the shared intermediate layer.

▶ **Theorem 30** (Vertical Compositionality). *If* $L_1 \vdash M_1 : L_2$ *and* $L_2 \vdash M_2 : L_3$*, then* $L_1 \vdash M_1 \triangleleft M_2 : L_3$.

As in the horizontal case, vertical compositionality relies on preservation of trace refinement under vertical composition. Since vertical composition consists of synchronization followed by hiding, this preservation reduces to preservation under synchronization (Lemma 31) and under hiding (Lemma 32); we omit the proof of the latter as it is straightforward.

▶ **Lemma 31.** *Given* $L, L' : \phi \to A$ *and* $M : A \to B$*, if* $\mathtt{sc}\, L \sqsubseteq \mathtt{sc}\, L'$*, then* $L \overline{\|} M \sqsubseteq L' \overline{\|} M$.

▶ **Lemma 32.** *Given* $L, L' : \phi \to A$ *and* $M : A \to B$*, if* $L \overline{\|} M \sqsubseteq L' \overline{\|} M$*, then* $L \mathbin{\overline{\triangleleft}} M \sqsubseteq L' \mathbin{\overline{\triangleleft}} M$.

The proof of Lemma 31 parallels the horizontal case. Although synchronization introduces more interaction than horizontal composition, executions can still be projected from, or combined into, those of the synchronized system.

▶ **Lemma 33** (Vertical projection). *For* $L : \phi \to A$*,* $M : A \to B$*, and* es*, if* $(s_1, s_2) \xrightarrow{es}_* (s_1', s_2')$ *is an execution in* $L \overline{\|} M$*, then* $s_1 \xrightarrow{\circ_A es}_* s_1'$ *in* $\mathtt{sc}\, L$ *and* $s_2 \xrightarrow{es}_* s_2'$ *in* $\mathtt{sc}\, M$.

▶ **Lemma 34** (Vertical combination). *For* $L : \phi \to A$*,* $M : A \to B$*, and* es*, if* $s_1 \xrightarrow{\circ_A es}_* s_1'$ *in* $\mathtt{sc}\, L$*,* $s_2 \xrightarrow{es}_* s_2'$ *in* $\mathtt{sc}\, M$*, and* $s_1.h | s_2.h \vdash_v es$*, then* $(s_1, s_2) \xrightarrow{es}_* (s_1', s_2')$ *in* $L \overline{\|} M$.

We omit vertical consistency ($m_1 | m_2 \vdash_v es$), as it mirrors the horizontal case. Vertical projection and combination ensure that synchronization preserves refinement (Lemma 31).

7 Evaluation and Discussion

Our Rocq development took about 9 person-months. The entire framework consists of 16.1k lines of code (LOC). It takes 1.7k LOC to formalize basic data types (e.g., maps), 1.3k LOC to formalize transition systems, the sequential consistency operator and composition operators as discussed in §4, 1.8k LOC to develop the refinement and forward/backward simulation as described in §5, and 11k LOC for horizontal and vertical compositionality as discussed in §6. As for the examples, we wrote proofs manually for them in Rocq. We wrote 5.5k LOC for the Register-Counter-Timer example, and 53.6k LOC for the bounded queue example. For the latter, it takes 7.4k LOC for basic definitions of array and the skeleton of simulation proofs, and 6.2k LOC for proving sequential consistency of these LTS to establish verified concurrent layers. The remaining 40k are for proving that a set of mutually dependent invariants hold for `ENQ` (20k) and `DEQ` (20k). Note that the proof for `ENQ` is almost identical to that for `DEQ` (with mostly copy and paste) thanks to their symmetric structures.

The sizes of proofs for examples are primarily influenced by two factors: the size of the system and the number of invariants that must be established. The former determines how many transition steps need to be considered in the inductive reasoning, while the latter determines how many times such inductive reasoning must be carried out. Consequently, the proof size can be approximated as $\#LOC \approx (\#invariants) \times (\#system\text{-}steps) \times p$, where p denotes the average number of lines of code required to prove that a single invariant is preserved by a single step. As a concrete example, consider the bounded queue. Its correctness relies on 12 mutually dependent invariants and 160 auxiliary independent invariants, yielding a total of 172 invariants. The system has 65 distinct transition steps. The Rocq proofs for this example consist of approximately 40K lines of code. Using the above approximation, this yields an average of $p = (40000/172/65) = 3.57$ lines of code per invariant per step.

With the above case studies, we show that foundational and compositional proofs for layered concurrent objects are *feasible*, even for non-trivial *lock-free* data structures with *only brute-force and manual* construction of Rocq proofs. In contrast, the existing denotational approach has only been tested against a basic queue using ticket locks *on paper* [46]. As discussed in §2.5, the verification in the denotational approach is actually carried out in an operational fashion using a program logic. The denotational approach requires defining functions that translate traces to states, and repeatedly performing case analyses over these functions at each proof step, which introduces complexity proportional to the number of translation cases. Analogous to the approximation of proof size for our approach given earlier in this section, we estimate the proof size of the denotational approach as $\#LOC' \approx (\#invariants) \times (\#system\text{-}steps) \times (\#translation\text{-}cases) \times p'$, where p' is the average number of lines required to prove a single case, and $\#translation\text{-}cases$ counts the average number of case analyses introduced by translation functions. Consequently, the proof size grows proportionally with both the number of system transition steps and the number of translation cases. For a more detailed comparison of proof complexity between our approach and the denotational one, we refer the reader to the extended version of the paper.

Of course, there is a very high amount of overhead to write proofs manually in Rocq as our examples show. To reduce this overhead, we plan to integrate our framework with automatic verification techniques for concurrent programs, which have been actively investigated. More specifically, we will investigate the connection between our LTS and TLA [37] in which correctness is also described as trace refinement and compositional inductive invariant inference seems promising [10]. We also plan to connect with simulation frameworks with built-in support for automation such as Veil [49].

We choose the lock-free bounded queue as our primary case study because it is not only a representative and non-trivial concurrent data structure for illustrating compositional

verification, but also a fundamental building block of real-world concurrent systems. For the former, the original proof our work is based upon verifies the queue as a complete system built directly over low-level `Register` and `Array` primitives [8]. In that work, there is no abstraction of `Counter` objects; operations such as `GET` and `INC` in Figure 17 correspond respectively to `READ` and `CAS` on `Register` objects (e.g., `rear.INC()` at line 7 of `ENQ` corresponds to `rear.CAS(r, r+1)`). Our compositional verification identifies and abstracts this pattern into a `Counter` object, thereby eliminating the need to reason explicitly about CAS operations and the associated low-level invariants when establishing the correctness of queues. For the latter, the LMAX Disruptor [58], a framework for low-latency inter-thread communication widely used in high-frequency trading platforms, can be understood as a composition of two concurrent bounded queues together with a component that processes data passed from one queue and then passes the result to another. As such, we could verify this system by composing our proofs with further refinements. We plan to investigate such real-world concurrent systems in the future to demonstrate the practical benefits of our approach.

In this work, we focus on verification of non-recursive program structures. For circular data structures such as trees or graphs, our proof approach can be applied when the goal is to verify that a concrete implementation is correct w.r.t a functional specification. For example, $L_{ary} \vdash M_{tree} : L_{tree}$ denotes the correctness of an array-based tree. In such cases, the verification proceeds similarly to our queue example by establishing simulation across layers. More challenging scenarios arise when recursive decomposition is required for circular data structures. For example, to verify a binary tree, we may show $L_{tree} \otimes L_{tree} \vdash M_{tree} : L_{tree}$ where the first two L_{tree} denote the left and right sub-trees, respectively, and M_{tree} uses them to implement a complete tree. This kind of proofs would require fixed-point reasoning and new proof principles for composing trace refinements. We conjecture that this extension is feasible, as our projection and combination theorems rely only on non-interference between objects and do not inherently depend on whether the implementation is circular or not.

8 Related Work and Conclusion

We discuss related work on compositional verification of *concurrent* programs. We elide a discussion of compositional verification of sequential programs (DeepSpec [21], FSCQ [7], ConFrm [25], etc.) which has been done in the introduction.

Layered Refinement. Existing approaches to verification of concurrent programs offer various degrees of compositionality. A first class of work treats programs as *closed* systems [26, 13, 12, 14, 41]. In these frameworks, programs are verified as a monolithic piece of code, which often results in proofs that only work for this particular program and cannot be further reused. A second category of work decomposes programs into libraries and their clients, so that the verified libraries can be linked with different clients to form verified closed programs [15, 53, 32, 60, 4, 20]. A particular example is verification of linearizability through observational refinement [15]. In essence, these frameworks support verification of closed programs consisting of *only two layers*: one at the bottom (libraries) and the other at the top (clients). A third category of work generalizes the above approach to an *arbitrary number of layers* of concurrent programs (e.g., CCAL [23], Civl [35], Compositional Linearizability [46], etc.), which enables building concurrent libraries out of existing libraries. For example, CCAL represents each abstraction layer as a transition system augmented with a global log recording all events at that layer; layers at different abstraction levels can be vertically composed. Civl models each layer of abstraction using structured programs, providing a

notion of layered refinement. Our work lies in the third category: we model both objects and implementations as transition systems equipped with layered interfaces that specify interactions across layers. A key distinction between our model and existing ones is that our verified layered objects do not expose invariants about internal states and running threads. This design significantly improves the modularity of verified layers, which enables extension of layers (i.e., horizontal compositionality), which we compare next.

Extension of Verified Layers. In multi-layer verification frameworks, although it is easy to build new layers on top of existing ones, it is quite difficult to extend an existing layer when concurrency is involved. For instance, CCAL allows horizontal composition only between two verified layers that share a common lower layer and global invariant [23]. A similar limitation appears in Civl, which introduces *yield invariants* [35] to reason about thread interference in the global state. In contrast, our work follows the approach of ConFrm [25] by taking objects – rather than threads – as the basic unit of a layer, and formalizes each object as an LTS that fully encapsulates all interference between threads on the object's state. Because of this encapsulation, our object layers can be horizontally extended easily.

Foundational Verification. In concurrent settings, encapsulation using formalized concurrent objects already occurs in the compositional theory of linearizability [46]. However, foundational verification in this framework has not yet been demonstrated, as discussed in §2.5. To address this problem, our work draws inspiration from *operational game semantics* [39] which establishes connections between transition systems and game semantics in sequential settings, and from I/O automata [42] which models transition systems in concurrent settings. These ideas collectively form the design of our concurrent LTS, which enables direct adoption of simulation techniques for I/O automata (and transition systems in general), thereby recovering the capability of foundational verification.

Concurrency Models. This work assumes preemptive concurrent semantics in the setting of sequential consistency, i.e., arbitrary interleaving of concurrent executions. This assumption is more relaxed than Civl which assumes cooperative semantics where concurrency occurs only at yield points [35]. Similar to TLA [37], our LTS directly encode the *native* behaviors of concurrent programs . This is more general than compositional linearizability which encodes concurrent interleaving of *sequential* processes and is specifically designed for *linearizable* objects. On the other hand, our work is more restrictive than existing work on two-level (library-client) layered refinement that can handle weak memory models [32, 53, 50]. One way to address this issue is to integrate operational weak memory semantics such as promising semantics [30, 38] into our framework, which is left for future work.

Automation. A limitation of our framework is the lack of automation: verification of non-trivial objects requires substantial manual effort, primarily in establishing invariants. Prior work has made progress on automatic invariant generation. Civl automates inductive invariants but does not produce proof certificates [35]. Other foundational frameworks that do not decompose programs into layers also provide automation; for example, Diaframe [44] automates linearizability proofs atop Iris [48, 29], and Veil [49, 19] automatically derives simple invariants and generates proofs in Lean [16]. Integrating similar automation techniques into our framework is an important direction for future work. As an initial effort, we have examined systems that can be automatically verified by Veil and analyzed the structure of invariants in their proofs. We observe that invariants that can be automated typically express properties about local states preserved by a single transition, without quantification over other states, threads, or executions. Based on this observation, we inspected the dozen

mutually dependent invariants used in the backward simulation proof of the bounded queue. We found that only three of them fall outside this category of invariants; they are for reasoning about multiple threads or relationships between states across multiple steps. For these invariants, manual proofs still seem necessary. As such, we plan to explore combination of automated and manual verification techniques to reduce the overall proof effort.

Concurrent Separation Logics. Concurrent separation logics such as VST [2], Iris [29] and their various extensions [62, 59, 55, 52, 17] could support general compositionality (beyond layered composition), foundational verification (e.g., embedded in Rocq or other theorem provers), preemptive and weak memory models, and support various automation techniques. However, their compositionality is designed for constructs in programming languages. Although such design enables powerful reasoning about intricate programs (e.g., dangling pointers), they are also at a lower level, tied to particular programming languages and limited in scalability. As such, we believe that they are more suitable for foundational verification of concrete programs, and will be very helpful for connecting our transition systems with actual code written in programming languages.

Conclusion. We have presented an approach to the compositional verification of layered concurrent programs via object-based transition systems with layered interfaces, which overcomes the the limitations of existing work in both compositionality and foundational verification. In the future, we plan to explore automation techniques that reduce the manual effort, and to integrate program logics to connect our transition systems with actual code. We also aim to extend our transition systems to support compositional *liveness* verification by incorporating relevant verification techniques [54, 61].

References

1 Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991. doi:10.1016/0304-3975(91)90224-P.

2 Andrew Appel. Verified software toolchain. In Gilles Barthe, editor, *Proc. 20th European Symposium on Programming (ESOP'11)*, volume 6602 of *LNCS*, pages 1–17. Springer, Saarbrucken, Germany, 2011. doi:10.1007/978-3-642-19718-5_1.

3 Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proc. 4th Symp on Formal Methods for Components and Objects*, 2005. doi:10.1007/11804192_17.

4 Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Concurrent library correctness on the tso memory model. In Helmut Seidl, editor, *Programming Languages and Systems*, pages 87–107, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-28869-2_5.

5 Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Argosy: verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1054–1068, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3314585.

6 Nicolas Chappe. A family of sims with diverging interests. *Proc. ACM Program. Lang.*, 10(POPL), January 2026. doi:10.1145/3776714.

7 Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 18–37, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2815400.2815402.

8 Robert Colvin and Lindsay Groves. Formal verification of an array-based nonblocking queue. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer*

Systems, ICECCS '05, pages 507–516, USA, 2005. IEEE Computer Society. doi:10.1109/ICECCS.2005.49.

9 Zhenyang Dai, Shuang Liu, Vilhelm Sjoberg, Xupeng Li, Yu Chen, Wenhao Wang, Yuekai Jia, Sean Noble Anderson, Laila Elbeheiry, Shubham Sondhi, Yu Zhang, Zhaozhong Ni, Shoumeng Yan, Ronghui Gu, and Zhengyu He. Verifying rust implementation of page tables in a software enclave hypervisor. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, pages 1218–1232, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3620665.3640398.

10 Ian Dardik and Eunsuk Kang. Compositional inductive invariant inference via assume-guarantee reasoning, 2025. doi:10.48550/arXiv.2509.06250.

11 John Derrick and Eerke Boiten. *Labeled Transition Systems and Their Refinement*, pages 3–26. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-92711-4_1.

12 John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Mechanizing a correctness proof for a lock-free concurrent stack. In Gilles Barthe and Frank S. de Boer, editors, *Formal Methods for Open Object-Based Distributed Systems*, pages 78–95, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-68863-1_6.

13 Brijesh Dongol and John Derrick. Verifying linearisability: A comparative survey. *ACM Comput. Surv.*, 48(2), September 2015. doi:10.1145/2796550.

14 Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *Proc. TACAS'10*, pages 296–311, 2010. doi:10.1007/978-3-642-12002-2_25.

15 Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010. doi:10.1016/j.tcs.2010.09.021.

16 Lean FRO. The Lean language, 2025. URL: https://lean-lang.org/.

17 Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. doi:10.1145/3498689.

18 Dan R. Ghica and Andrzej S. Murawski. Angelic semantics of fine-grained concurrency. *Annals of Pure and Applied Logic*, 151(2-3):89–114, 2008. doi:10.1016/j.apal.2007.10.005.

19 Vladimir Gladshtein, George Pîrlea, Qiyuan Zhao, Vitaly Kurin, and Ilya Sergey. Foundational multi-modal program verifiers. *Proc. ACM Program. Lang.*, 10(POPL), January 2026. doi:10.1145/3776719.

20 Alexey Gotsman and Hongseok Yang. Liveness-preserving atomicity abstraction. In *Proceedings of the 38th International Conference on Automata, Languages and Programming - Volume Part II*, ICALP'11, pages 453–465, Berlin, Heidelberg, 2011. Springer-Verlag. doi:10.1007/978-3-642-22012-8_36.

21 Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan(Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In Sriram K. Rajamani and David Walker, editors, *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*, pages 595–608, New York, NY, USA, 2015. ACM. doi:10.1145/2775051.2676975.

22 Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 653–669, GA, 2016. USENIX Association. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

23 Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjober, Hao Chen, David Costanzo, and Tahnia Ramananandro. Certified concurrent abstraction layers. In Jeffrey S. Foster and Dan Grossman, editors, *Proc. 2018 ACM Conference on*

Programming Language Design and Implementation (PLDI'18), pages 646–661, New York, NY, USA, 2018. ACM. doi:10.1145/3192366.3192381.

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

25 Atalay Mert Ileri, Nickolai Zeldovich, Adam Chlipala, and Frans Kaashoek. Probability from possibility: Probabilistic confidentiality for storage systems under nondeterminism. In *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*, pages 96–111, 2024. doi:10.1109/CSF61375.2024.00041.

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

27 Cliff B Jones. Specification and design of (parallel) programs. In *9th IFIP World Computer Congress (Information Processing 83)*. Newcastle University, 1983. URL: https://api.semanticscholar.org/CorpusID:38308402.

28 Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 256–269, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2951913.2951943.

29 Ralf Jung, David Swasey, Filip Sieczkowski, Ksper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*, pages 637–650, 2015. doi:10.1145/2775051.2676980.

30 Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proc. 44th ACM Symposium on Principles of Programming Languages (POPL'17)*, pages 175–189, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009850.

31 Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew Parkinson. Proving linearizability using partial orders. In Hongseok Yang, editor, *Programming Languages and Systems*, pages 639–667, Heidelberg, 2017. Springer Berlin Heidelberg. doi:10.1007/978-3-662-54434-1_24.

32 Artem Khyzha and Ori Lahav. Abstraction for crash-resilient objects. In Ilya Sergey, editor, *Programming Languages and Systems*, pages 262–289, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-030-99336-8_10.

33 G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, et al. seL4: Formal verification of an OS kernel. In *SOSP'09*, pages 207–220, October 2009. doi:10.1145/1629575.1629596.

34 Bernhard Kragl and Shaz Qadeer. Layered concurrent programs. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 79–102, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-96145-3_5.

35 Bernhard Kragl and Shaz Qadeer. The civl verifier. In *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 143–152, 2021. doi:10.34727/2021/isbn.978-3-85448-046-4_23.

36 Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. Refinement for structured concurrent programs. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 275–298, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-53288-8_14.

37 Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994. doi:10.1145/177492.177726.

38 Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. Promising 2.0: Global optimizations in relaxed memory concurrency. In *Proc. 2020 ACM Conference on Programming Language Design and Implementation (PLDI'20)*, pages 362–376, New York, NY, USA, 2020. ACM. doi:10.1145/3385412.3386010.

39 Paul Blain Levy and Sam Staton. Transition systems over games. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on*

Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014, pages 64:1–64:10. ACM, 2014. doi:10.1145/2603088.2603150.

40 Xupeng Li, Xuheng Li, Wei Qiang, Ronghui Gu, and Jason Nieh. Spoq: Scaling {Machine-Checkable} systems verification in coq. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 851–869, 2023. URL: https://www.usenix.org/conference/osdi23/presentation/li-xupeng.

41 Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. Model checking linearizability via refinement. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods*, pages 321–337, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-05089-3_21.

42 Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI quarterly*, 2:219–246, 1989. URL: https://api.semanticscholar.org/CorpusID:10292247.

43 Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. Untimed systems. *Inf. Comput.*, 121(2):214–233, 1995. doi:10.1006/inco.1995.1134.

44 Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: automated verification of fine-grained concurrent programs in iris. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 809–824, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519939.3523432.

45 Peter W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR'04*, pages 49–67, 2004. doi:10.1016/j.tcs.2006.12.035.

46 Arthur Oliveira Vale, Zhong Shao, and Yixuan Chen. A compositional theory of linearizability. *J. ACM*, 71(2), April 2024. doi:10.1145/3643668.

47 Susan Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, May 1976. doi:10.1145/360051.360224.

48 Sunho Park, Jaewoo Kim, Ike Mulder, Jaehwang Jung, Janggun Lee, Robbert Krebbers, and Jeehoon Kang. A proof recipe for linearizability in relaxed memory separation logic. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi:10.1145/3656384.

49 George Pîrlea, Vladimir Gladshtein, Elad Kinsbruner, Qiyuan Zhao, and Ilya Sergey. Veil: A framework for automated and interactive verification of transition systems. In Ruzica Piskac and Zvonimir Rakamarić, editors, *Computer Aided Verification*, pages 26–41, Cham, 2025. Springer Nature Switzerland. doi:10.1007/978-3-031-98682-6_2.

50 Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290381.

51 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74, 2002. doi:10.1109/LICS.2002.1029817.

52 Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. Islaris: verification of machine code against authoritative isa semantics. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 825–840, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519939.3523434.

53 Abhishek Kr Singh and Ori Lahav. An operational approach to library abstraction under relaxed memory concurrency. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571246.

54 Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. Transfinite iris: resolving an existential dilemma of step-indexed separation logic. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 80–95, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454031.

55 Simon Spies, Niklas Mück, Haoyi Zeng, Michael Sammler, Andrea Lattuada, Peter Müller, and Derek Dreyer. Destabilizing iris. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025. doi:10.1145/3729284.

56 Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 866–881, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483560.

57 Rocq Development Team. The rocq prover. version 9.0.0, released March 12, 2025., 2025. URL: https://rocq-prover.org/.

58 Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. Disruptor: High performance alternative to bounded queues for, 2011.

59 Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. Trillium: Higher-order concurrent and distributed separation logic for intensional refinement. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi:10.1145/3632851.

60 Xiaoxiao Yang, Joost-Pieter Katoen, Huimin Lin, Gaoang Liu, and Hao Wu. Branching bisimulation and concurrent object verification. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 267–278, 2018. doi:10.1109/DSN.2018.00037.

61 Qiyuan Zhao, George Pîrlea, Karolina Grzeszkiewicz, Seth Gilbert, and Ilya Sergey. Compositional verification of composite byzantine protocols. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, pages 34–48, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3658644.3690355.

62 Litao Zhou, Jianxing Qin, Qinshi Wang, Andrew W. Appel, and Qinxiang Cao. Vst-a: A foundationally sound annotation verifier. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi:10.1145/3632911.