Foundational and Compositional Verification of Layered Concurrent Objects (Artifact)

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

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

Abstract

This document provides artifact description for the ECOOP 2026 paper "Foundational and Compositional Verification of Layered Concurrent Objects".

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

Keywords and phrases Formal Verification, Concurrency, Formalization, Rocq

Digital Object Identifier 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.

Related Article Yicheng Ni and Yuting Wang, "Foundational and Compositional Verification of Layered Concurrent Objects", in 40th European Conference on Object-Oriented Programming (ECOOP 2026), LIPIcs, Vol. 372, pp. 21:1–21:31, 2026.
`https://doi.org/10.4230/LIPIcs.ECOOP.2026.21`

Related Conference 40th European Conference on Object-Oriented Programming (ECOOP 2026), June 29–July 3, 2026, Brussels, Belgium

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2026 Call for Artifacts and the ACM Artifact Review and Badging Policy.

1 Scope

This artifact supports the formalization and verification claims of the paper "Foundational and Compositional Verification of Layered Concurrent Objects". In particular, the experimental claims made in the paper are:

- the complete development, including all definitions, lemmas, and theorems from Sections 4, 5, and 6, fully formalized in Rocq;
- the reported proof effort, measured as LOC (lines of code) for each component, as described in Section 7.

All these claims are fully mechanized and can be validated by compiling the Rocq development and reproducing the proof effort statistics. Instructions for a quick check are provided in Section 3, while a full verification procedure is described in the Appendix.

2 Content

The artifact package includes the following components:

[1] Corresponding author

- **Rocq formalization**
  - Type: proof artifact
  - Format: Rocq/Coq source files (.v)
  - Location: `src/`
  - Description: The complete mechanized development of the paper, including (For a complete description of the source code layout, please refer to Appendix B):
    - basic data structures and libraries,
    - semantic model,
    - correctness of verified concurrent objects,
    - compositional verification framework,
    - verified examples.
- **Build system**
  - Type: code
  - Format: Makefile and build scripts
  - Location: root directory of the project
  - Description: push-button scripts to compile the source code and reproduce the proof efforts.
- **Documentation**
  - Type: documentation
  - Format: README.md
  - Location: root directory of the project
  - Description: Detailed description of the artifact structure, correspondence with the paper, and instructions for building and evaluation.

Open-source and reproducibility. The implementation is fully included in the artifact, both in the Zenodo Docker image and in the GitHub repository (see Section 3). It can be recompiled by following the instructions provided in Appendix D.

Reuse scenarios.

- The concurrent objects verified in this artifact can be further used to build higher-level objects.
- The compositional verification framework can be reused for verifying other concurrent systems.
- The compositional verification framework can be extended (e.g., by defining new transition systems or composition operators) to verify concurrent programs with richer structures.

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). In addition, the artifact is also available at:

- zenodo (as Docker image) (`https://doi.org/10.5281/zenodo.19876027`)
- github (as source code) (`https://github.com/Sam-Ni/verified-concurrent-objects-ecoop2026-artifact`)

To quickly test the artifact with Docker image, proceed as follows:

1. Download the Docker image named `vco-ecoop-artifact.tar.gz`.
2. Load the Docker image:

```
docker load -i vco-ecoop-artifact.tar.gz
```

3. Run the container:

```
docker run -it --rm vco-ecoop-artifact
```

4. Compile the development (`-j4` is optional):

```
make -j4
```

5. Reproduce proof efforts:

```
make statistics
```

Expected result:

1. By running `make -j4`, all Rocq files compile successfully without errors. Any warnings during compilation are due to deprecations in Rocq versions, as the code was originally developed with Rocq 8.19.1 but is compiled here using Rocq 8.20.1 to improve compilation performance.
2. By running `make statistics`, the LOC for each component of the project and a final summary will be printed.

The compilation checks all definitions, lemmas, and theorems described in the paper. The statistics checks the proof efforts claimed in the paper.

4 Tested platforms

Please see the metadata in HotCRP.

5 License

The artifact is available under the MIT license.

6 MD5 sum of the artifact

0f4ea2fa9488552e2d47ec07bc579be6

7 Size of the artifact

1.57 GB

A Overview of the Appendix

We first introduce the structure of this artifact in Appendix B. The formal definitions and theorems of the claims we made in the submission paper can be found in Appendix C. The instructions for building and evaluating can be found in Appendix D and Appendix E.

B Structure of the Artifact

The source code is located in the `src` directory and is organized into five parts: (1) basic data types and properties, (2) the semantic model, (3) correctness of verified concurrent objects, (4) compositionality, and (5) examples. Below, we describe the functionality of each file following this structure. A complete correspondence between the formal definitions, lemmas, and theorems in the artifact and those in the paper is provided in Appendix C.

Basic data types and properties

- LibVar.v : encode the set of processes P (bijective to nat).
- LibEnv.v : encode functions from P to other types as lists, define operations to modify these functions (e.g., remove and substitute) and prove properties for the operations.
- LibTactic.v : Tactics to automate proofs in LibEnv.v.
- VeriTactics.v : simple Tactics to destruct match expressions, used in inductive steps.

Semantic model

- LTS.v : define interfaces, labelled transition system, synchronized system (which corresponds to Section 4.1 in the paper). Also define event, execution and trace (Section 5.1).
- SyncLTS.v : define sequential consistency operation (Section 4.2).
- Tensor.v : define horizontal composition (Section 4.3).
- Compose.v : define synchronization (Section 4.4).
- Link.v : define hiding (Section 4.4).
- Invariants.v : define and establish invariants about transition systems.
- ComposedLTS.v : define and establish invariants about synchronized systems.
- KInduction.v : define step-indexed induction principle of LTS, used in the proof of invariant of the bounded queue.

Correctness of verified concurrent objects

- Refinement.v and ImplRefinement.v : define trace refinement between transition systems (Section 5).
- VerifiedConcurrentObject.v : define verified concurrent object (Section 5, Definition veriobj corresponds to Definition 17).
- BSim.v : define backward simulation between synchronized systems and plain transition systems. Derive this simulation implies trace refinement.
- ComposedRefinement.v : define forward simulation between synchronized systems and plain transition systems. Derive this simulation implies trace refinement.

Compositionality

- RawTensor.v : define plain horizontal composition (Section 6.1).
- RawCompose.v : define plain synchronization (Section 6.1).
- HE.v : prove horizontal extraction (Section 6.1, Lemma tensor_raw_tensor corresponds to Proposition 24).
- SyncCompLTS.v : define sequential consistency operator for synchronized systems (Section 6.1).
- VE.v : prove vertical extraction (Section 6.1, Theorem sync_raw_compose_refines_compose corresponds to Proposition 25).
- Tensor.v : prove horizontal preservation (Section 6.2, Theorem tensor_preserves_refines corresponds to Lemma 27).
- ImplTensor.v : define horizontal composition between implementation, used in HComp.v.
- HComp.v : prove horizontal compositionality (Section 6.2, Theorem HComp corresponds to Theorem 26).
- Compose.v : prove synchronization preserves trace refinement (Section 6.3, Lemma refines_composed_refines corresponds to Lemma 31).
- Link.v : prove hiding retains trace refinement and prove vertical preservation (Section 6.3, Lemma linked_refines_congruence corresponds to Lemma 32).

- ImplRawCompose.v : define vertical composition between implementations, used in VComp.v.
- VComp.v : prove vertical compositionality (Section 6.3, Theorem VComp corresponds to Theorem 30).
- Linking.v : prove proof rule LINK (Fig. 16 in Section 3.3, Theorem Link corresponds to rule LINK).
- Weaken.v : prove proof rule WEAKEN (Fig. 16 in Section 3.3, Theorem Weaken corresponds to rule WEAKEN).

Examples

Register-Counter-Timer

- Register.v : formalize the Register object (Figure 9 in Section 3.1 and Example 3 in Section 4.1).
- RegisterCounterImpl.v : formalize the Counter implementation (Figure 9 in Section 3.1 and Example 4 in Section 4.1).
- CounterTimerImpl.v : formalize the Timer implementation (Figure 9 in Section 3.1).
- Counter.v : formalize the Counter specification (CounterSpec) (Figure 15 in Section 3.2).
- Timer.v : formalize the Timer specification (TimerSpec).
- RegisterCounter.v : define the synchronization of Register object and Counter implementation (Example 9 in Section 4.4).
- RegisterCounterCorrect.v : prove Counter implemented by Register refines CounterSpec through establishing forward simulation (Theorem register_counter_correct corresponds to the informal illustration in Section 3.2).
- TickNat.v : define the mapping between nat and time (hour, minute), used to prove simulation of CounterTimer.
- CounterTimerCorrect.v : prove Timer implemented by CounterSpec refines TimerSpec through establishing forward simulation (Theorem counter_timer_correct).
- RegisterTimer.v : prove Timer implemented by Counter implemented by Register refines TimerSpec (Theorem register_timer_correct corresponds to the statement in the second paragraph in Section 3.4).
- other files with prefix “Register” or “Counter” or “Timer” : define and prove properties about these systems.

Array-based queue

- Array.v : formalize the Array object (Figure 17).
- ArrayQueueImpl.v : formalize the Queue implementation (Figure 17).
- Queue.v : formalize the Queue specification (QueueSpec) (Figure 18).
- ArrayQueue.v : define the horizontal composition of Array with two Counters. Then define the vertical composition of these systems with the queue implementation (the component of equation (1) in Section 3.4).
- Identity.v : define the “copycat” implementation used to horizontally compose Array with two counters (M_{id} in section 3.4).
- Files with prefix “ArrayQueueInvariant”: define and prove invariants about the ArrayQueue.
- AQC.v : prove the ArrayQueue refines QueueSpec by establishing backward simulation (Example 20 in Section 5).
- QC.v : prove the correctness of the whole queue (The proof corresponds to the derivation in Section 3.4, with Lemma whole_queue_correctness corresponding to equation 5 in Section 3.4).
- other files with prefix “Array” or “Queue”: define and prove properties about these systems.

C List of Claims

We list the definitions, lemmas and theorems from each section of the submission paper below along with the references to their corresponding Rocq formalization in the artifact. Note that the Rocq definitions and file names are **clickable**, allowing you to navigate directly to their locations in the GitHub repository.

C.1 Section 3

- The VCOMP proof rule from Section 3.3 (line 463 and Figure 16) corresponds to VComp in the Rocq file src/VComp.v.
- The HCOMP proof rule from Section 3.3 (line 467 and Figure 16) corresponds to HComp in the Rocq file src/HComp.v.
- The LINK proof rule from Section 3.3 (line 471 and Figure 16) corresponds to Link in the Rocq file src/Linking.v.
- The WEAKEN proof rule from Section 3.3 (line 471 and Figure 16) corresponds to Weaken in the Rocq file src/Weaken.v.
- The statement of verified bounded queue from Section 3.4 (line 510, equation 1) corresponds to verified_array_queue in the Rocq file src/QC.v.
- The statement of verified array horizontally composed with counters from Section 3.4 (line 517, equation 2) corresponds to verified_array_counter_counter in the Rocq file src/QC.v.
- The statement of verified queue vertically composed with registers from Section 3.4 (line 521, equation 3) corresponds to verified_whole_queue in the Rocq file src/QC.v.
- The statement of verified queue applied LINK and WEAKEN rules from Section 3.4 (line 523, equation 4) corresponds to verified_queue_link_and_weaken in the Rocq file src/QC.v.
- The statement of trace refinement for the verified queue from Section 3.4 (line 525, equation 5) corresponds to whole_queue_correctness in the Rocq file src/QC.v.

C.2 Section 4

- Definition 1 from Section 4.1 (line 533) is defined as language_interface in the Rocq file src/LTS.v.
- Definition 2 from Section 4.1 (line 535) is defined as lts in the Rocq file src/LTS.v.
- Example 3 from Section 4.1 (line 553) is defined as Register in the Rocq file src/Register.v.
- Example 4 from Section 4.1 (line 557) is defined as register_counter_impl in the Rocq file src/RegisterCounterImpl.v.
- Definition 5 from Section 4.2 (line 571) is defined as sync in the Rocq file src/SyncLTS.v.
- Definition 6 from Section 4.3 (line 584) is defined as tensor in the Rocq file src/Tensor.v.
- Definition 7 from Section 4.4 (line 611) is defined as composed_lts in the Rocq file src/LTS.v.
- Definition 8 from Section 4.4 (line 619) is defined as compose in the Rocq file src/Compose.v.
- Example 9 from Section 4.4 (line 630) is defined as composed_register_counter in the Rocq file src/RegisterCounter.v.
- Definition 10 from Section 4.4 (line 638) is defined as linked_lts in the Rocq file src/LTS.v.
- Definition 11 from Section 4.4 (line 643) is defined as $\triangleleft\!-'$ in the Rocq file src/Link.v.

C.3 Section 5

- Definition 12 from Section 5 (line 652) is defined as event in the Rocq file src/LTS.v.
- Definition 13 from Section 5 (line 654) is defined as valid_execution_fragment in the Rocq file src/LTS.v.

- Definition 14 from Section 5 (line 657) is defined as in_trace in the Rocq file src/LTS.v.
- Definition 15 from Section 5 (line 659) is defined as impl_refines in the Rocq file src/ImplRefinement.v.
- Definition 16 from Section 5 (line 663) is not explicitly included in the artifact, as we always unfold it at the points where it is used (e.g., in the rule of LINK).
- Definition 17 from Section 5 (line 666) is defined as veriobj in the Rocq file src/VerifiedConcurrentObject.v.
- Definition 18 from Section 5 (line 673) is defined as composed_bsim_properties in the Rocq file src/BSim.v.
- Theorem 19 from Section 5 (line 686) is stated as backward_simulation in the Rocq file src/VerifiedConcurrentObject.v.
- Example 20 from Section 5 (line 693) is proved as array_queue_correct in the Rocq file src/AQC.v.

C.4 Section 6

- Definition 21 from Section 6.1 (line 721) is defined as tensor in the Rocq file src/RawTensor.v.
- Definition 22 from Section 6.1 (line 726) is defined as raw_compose in the Rocq file src/ImplRawCompose.v.
- Definition 23 from Section 6.1 (line 728) is defined as $\triangleleft$ in the Rocq file src/ImplRawCompose.v.
- Proposition 24 from Section 6.1 (line 731) is stated as tensor_raw_tensor in the Rocq file src/HE.v.
- Proposition 25 from Section 6.1 (line 733) is stated as sync_raw_compose_refines_compose in the Rocq file src/VE.v.
- Theorem 26 from Section 6.2 (line 741) is stated as HComp in the Rocq file src/HComp.v.
- Lemma 27 from Section 6.2 (line 745) is stated as tensor_preserves_refines in the Rocq file src/Tensor.v.
- Lemma 28 from Section 6.2 (line 753) is stated as valid_execution_fragment_com in the Rocq file src/Tensor.v.
- Lemma 29 from Section 6.2 (line 761) is stated as valid_execution_fragment_join in the Rocq file src/Tensor.v.
- Theorem 30 from Section 6.3 (line 770) is stated as VComp in the Rocq file src/VComp.v.
- Lemma 31 from Section 6.3 (line 776) is stated as refines_composed_refines in the Rocq file src/Compose.v.
- Lemma 32 from Section 6.3 (line 777) is stated as composed_refines_linked_refines in the Rocq file src/Link.v.
- Lemma 33 from Section 6.3 (line 781) is stated as valid_execution_fragment_com in the Rocq file src/Compose.v.
- Lemma 34 from Section 6.3 (line 783) is stated as valid_execution_fragment_join in the Rocq file src/Compose.v.

C.5 Section 7

- We claim that we wrote 15.7k lines of code (LOC) the entire framework, 5.4k LOC for the Register-Counter-Timer example, and 53.9k LOC for the bounded queue example in Section 7 (line 788, 794 and 795). The details are discussed in Appendix E below.

D Compilation

Requirements

The development is known to compile with Rocq v8.20.1 and OCaml v4.14.1.

- If you are using the docker image, all the required software have already been installed in the container.
- If you prefer to compile the source code on your own computer, then we recommend using the `opam` package manager to set up a build environment in Linux. We have tested the building on Linux with the following shell commands.

```
# Initialize opam (if you haven't used it before)
opam init

# Create an "opam switch" dedicated to building the code
opam switch create vco ocaml-base-compiler.4.14.1

# Configure the current shell to use the newly created opam switch
eval $(opam env --switch=vco)

# Install the Rocq (coq) packages
opam pin add coq 8.20.1
```

Instructions for compiling the Rocq code

Download the source code from github (If you are using the Docker image, ignore this):

```
git clone git@github.com:Sam-Ni/verified-concurrent-objects-ecoop2026-artifact.
    git
```

The Rocq code is located in the `src` directory. You can compile the project by running:

```
make
```

You are all set if the compilation finishes successfully. You may also speed up the process by using `-jN` argument to run the Coq compiler in parallel. We have tested running `make` in the docker, which in turn runs on a host machine with Intel Core Ultra 7 255H (16 cores) and 16 GB memory. The whole compilation takes 11 minutes. When using `make -4j` command for parallel compilation, it takes about 6 minutes.

You can run `make clean` to clean up the compiled code.

Navigating the proofs

- If you have compiled the source code on your local machine, you can use your preferred editor and plugins to explore and run the proofs interactively.
- If you are working within the Docker container, the environment provides `emacs` with the `proof-general` plugin for interactive navigation of Rocq files.
  For example, to open `AQC.v`, run:

  ```
  emacs -nw src/AQC.v
  ```

Within Emacs, you can step through the file interactively by pressing:

```
C-c C-n
```

E Evaluation

E.1 Soundness

To check that there is no admit in the artifact, run

```
find . -name "*.v" | xargs grep "Admitted"
```

which should show no admit.

E.2 Proof effort

The following instructions describe how to reproduce the lines of code (LOC) for each component of our implementation, as reported in Section 7 (lines 788–799). The reported LOC corresponds to the sum of specification and proof lines (i.e., the first two columns of the output by `coqwc`), excluding comments. The exact numbers may differ slightly from those in the submission, since some files contain code spanning multiple components. However, the total LOC remains consistent with the LOC reported in the submission.

E.2.1 Basic Data Types

Run the following command in directory `src`:

```
coqwc Lib*.v
```

The last row of the results should be

```
642    1088       0 total
```

Therefore, we used 1.7k ($1730 = 642 + 1088$) lines of code to formalize basic data types.

E.2.2 Definiton of Labelled Transition Systems

Run the following command in directory `src`:

```
coqwc LTS.v SyncLTS.v
```

The last row of result should be

```
785     522      35 total
```

We used 1.3k ($1307 = 785 + 522$) lines of code to formalize transition systems and the sequential consistency operator.

E.2.3 Trace Refinement, Simulation and Verified Concurrent Object

Run the following command in directory `src`:

```
coqwc BSim.v ComposedRefinement.v ImplRefinement.v Invariants.v Refinement.v \
VerifiedConcurrentObject.v ComposedLTS.v
```

The last row of result should be

```
566    1322       4 total
```

We used 1.8k ($1888 = 566 + 1322$) lines of code to define trace refinement, forward/backward simulation and verified concurrent objects.

E.2.4 Compositionality

Run the following command in directory `src`:

```
coqwc RawTensor.v RawCompose.v HE.v VE.v SyncCompLTS.v Tensor.v ImplTensor.v \
HComp.v Compose.v Link.v ImplRawCompose.v Linking.v Weaken.v VComp.v
```

The last row of result should be

```
2045    9192      67 total
```

We used 11k ($11237 = 2045 + 9192$) lines of code for horizontal and vertical compositionality with composition rules Link and Weaken.

E.2.5 The Entire Framework

The total LOC of the framework is computed as the sum of the LOC of the four components listed above: basic data types (1730), definition of labeled transition systems (1307), trace refinement, simulation, and verified concurrent objects (1888), and compositionality (11237).

```
1730 + 1307 + 1888 + 11237 = 16162
```

Therefore, the entire framework consists of 16.1k(16162) lines of code (LOC).

E.2.6 Register-Counter-Timer example

Run the following command in directory `src`:

```
coqwc Register*.v Counter*.v Timer.v TickNat.v
```

The last row of result should be

```
1417    4130      53 total
```

We used 5.5k ($5547 = 1417 + 4130$) lines of code to prove the Register-Counter-Timer example.

E.2.7 The bounded queue example

Run the following command in directory `src`:

```
coqwc A*.v Q*.v KInduction.v Identity.v VeriTactics.v
```

The last row of result should be

```
6998   46602    1156 total
```

Totally, we used 53.6k ($53600 = 6998 + 46602$) lines of code to prove the bounded queue example.

Below, we report the LOC for each component in the bounded queue example.

Basic definitions and the skeleton of simulation proofs

Run the following command in directory `src`:

```
coqwc AQC.v Array.v ArrayProp.v ArrayQueue.v ArrayQueueImpl.v \
ArrayQueueImplProp*.v ArrayQueueMapping.v ArrayQueueProp.v Queue.v \
KInduction.v VeriTactics.v
```

The last row of result should be

```
1776    5658     121 total
```

We used 7.4k ($7434 = 1776 + 5658$) lines of code for basic definitions and the skeleton of simulation proofs.

Establishing verified concurrent layers

Run the following command in directory `src`:

```
coqwc ArrayCorrectness.v ArraySC.v ArrayQueueImplSC.v ArrayQueueReduceSC.v \
ArrayToQueue.v QC.v QueueProp.v
```

The last row of result should be

```
592    5694     299 total
```

We used 6.2k ($6286 = 592 + 5694$) lines of code for proving sequential consistency of the LTS to establish verified concurrent layers.

Proof of invariants hold for the bounded queue

Run the following command in directory `src`:

```
coqwc ArrayQueueInvariant*.v Identity.v
```

The last row of result should be

```
4630   35250     736 total
```

Therefore, we used 40k ($39880 = 4630 + 35250$) lines of code for proving that a set of mutually dependent invariants hold for ENQ and DEQ of the bounded queue.