Verifying Wait-Freedom for Concurrent Higher-Order Programs (Artifact)

Egor Namakonov
Aarhus University, Denmark

Lars Birkedal
Aarhus University, Denmark

Amin Timany
Aarhus University, Denmark

Abstract

Wait-freedom is the strongest non-blocking progress guarantee for concurrent data structures, ensuring that every operation completes in a finite number of steps regardless of interference from other threads. While verification of wait-freedom has been studied for first-order languages, verifying it for higher-order programming languages with general references remains an open challenge. In such languages, operations may be used by arbitrary, unverified higher-order clients, making it unclear how to even define wait-freedom formally in terms of programs' semantics, let alone prove it.

In this paper, we present the first framework for verifying wait-freedom of concurrent programs written in a higher-order language with general references. Our approach is based on the Lawyer concurrent separation logic which has been recently introduced for termination verification. We identify a specification pattern in the Lawyer logic that captures wait-freedom. To establish this connection formally, we extend Lawyer with a novel adequacy theorem that proves that programs which are proven correct in the Lawyer logic against a specification in this aforementioned specification pattern are wait-free. Proving wait-freedom requires us to show that all calls made to operations by any arbitrary client terminate. Thus, as part of formally proving wait-freedom, *i.e.*, as part of the proof of the adequacy theorem above, we need to prove that the behavior of the client of the data structure is safe in the sense that it does not break the internal invariants of the data structure, *e.g.*, by directly manipulating the data structure's internal state. To this end, we develop a logical relations model that establishes safety for all clients once and for all.

We demonstrate the effectiveness of our approach by proving wait-freedom for several representative examples, including higher-order functions such as the list map function, and a memory-efficient single-producer, single-consumer queue. For the latter, wait-freedom is conditional in that as the name suggests there can be at most one enqueuer thread and one dequeuer thread. To capture this formally we introduce the notion of restricted wait-freedom as a variant of wait-freedom that restricts the number of concurrent threads, and show how our approach can support reasoning about restricted wait-freedom. All our results have been mechanized on top of the Rocq Prover and using the Iris separation logic framework that Lawyer is also based on.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Theory of computation → Logic and verification; Theory of computation → Concurrency

Keywords and phrases separation logic, higher-order logic, concurrency, formal verification

Digital Object Identifier 10.4230/DARTS.12.1.21

Funding This work was supported in part by a Villum Investigator grant VIL73403, Center for Basic Research in Program Verification (CPV, 25804), from Villum Fonden. This work was co-funded by the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

Related Article Egor Namakonov, Lars Birkedal, and Amin Timany, "Verifying Wait-Freedom for Concurrent Higher-Order Programs", in 40th European Conference on Object-Oriented Programming (ECOOP 2026), LIPIcs, Vol. 372, pp. 20:1–20:29, 2026.
https://doi.org/10.4230/LIPIcs.ECOOP.2026.20

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 contains the mechanization of the results presented in our paper "Verifying wait-freedom for concurrent higher-order programs" [2]. In particular, it presents an extension of the recent Lawyer project [3] for establishing wait-freedom. Moreover, it contains a virtual machine with the pre-built mechanization.

The mechanization includes all the results on wait-freedom presented in the paper, namely:

- Formalized definition of "wait-freedom" in Rocq. Following the paper, we also provide variations of the definitions, specifications and theorems to account for different progress properties.
- A specification format that internalizes wait-freedom in Lawyer logic;
- Adequacy theorem that reduces establishing wait-freedom to proving the aforementioned triple;
- Case studies on establishing (variations of) wait-freedom with this approach:
  - wait-freedom of `incr` (Figure 1) and a simple sequential program (not present in the paper);
  - possibly-stuck wait-freedom of list mapping function (Figure 6) specialized to `incr`;
  - restricted wait-freedom of a single-producer single-consumer queue (Figure 7).

See the detailed correspondence between the Rocq mechanization and the definitions from paper in the appendix.

2 Content

The artifact consists of the following:

- `wfree_suppl.zip`: Rocq mechanization of all the results in the paper;
- `wfree.ova`: VirtualBox VM with pre-built mechanization. Username/password: `lawyer`. All relevant files are located in `/home/lawyer/artifact`.
- `paper-appendix.pdf`: full version of the paper with appendix.

Most of our development is an extension of the Lawyer project. However, we also use a fork of Trillium, as we generalize its adequacy theorem and weakest precondition. The overall structure of the mechanization is as follows:

- `trillium/` – our fork of Trillium framework. The main changes are located in:
  - `bi/weakestpre.v` – generalization of weakest precondition parameterized with the bit that allows or prohibits forking
  - `program_logic/adequacy_cond.v` – definition of the progress resource and proof of conditional adequacy theorem
- `lawyer/nonblocking` – wait-freedom extension of Lawyer
  - `.` – most of wait-freedom development, except for restricted wait-freedom and logical relations
  - `tokens/` – adaptation to the restricted wait-freedom
  - `logrel/` – definitions and theorems about logical relations
  - `examples/` – case studies on wait-freedom

Reusability

The sources for our development are provided inside the VM and as a standalone `wfree_suppl.zip` file. See the appendix for the instructions for manual build.

Verification of further case studies should follow the approach outlined below:

1. Choose the appropriate progress property: wait-freedom, possibly-stuck wait-freedom, or restricted wait-freedom.
2. Prove the corresponding specification for the case study algorithm; in particular, provide an instance of `WaitFreeSpec` or `WaitFreeSpecToken`;
3. Apply the corresponding adequacy theorem.

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 DOI **10.5281/zenodo.19607892**.

For the kick-the-tires stage, we suggest to check the pre-built mechanization in VM as follows:

1. Setup the VM
   a. Install VirtualBox (we used the version 7.2.4)
   b. Download the `wfree.ova` file.
   c. Open VirtualBox and navigate to `File/Import Appliance`. Provide the path to the downloaded `wfree.ova` and follow instructions to create a VM.
2. Run the newly created VM. The rest of the instructions below are to be executed inside the VM.
3. Build the Lawyer project
   a. Navigate to `/home/lawyer/artifact/lawyer`.
   b. Run `make -j 4`. This will build the mechanization of Lawyer, including our wait-freedom extension. It should terminate in a second, as the proofs are already built.
4. Check that the wait-freedom of case studies is proven up to axioms.
   a. Open `check/check_wfree.v` with an editor of choice; the VM provides Emacs+Proof General and VSCode+vsrocq.
   In that file, the definition `wfree_results` is a tuple collecting the proofs of progress properties of all case studies from the paper and one extra.
   b. Step through every line of this file (`Next` button in Proof General or `Step Forward` button in vsrocq).
   c. The last `Print Assumptions` line prints all the axioms that the listed theorems rely on.
   Processing it might take a while, and vsrocq might appear to be stuck; wait for up to a minute for it to complete.
   The used axioms will be listed at the bottom of the output. Check that only the following axioms are used:
   - `RelationalChoice.relational_choice`
   - `ClassicalUniqueChoice.dependent_unique_choice`
   - `Classical_Prop.classic`
   - `classical.PropExt`
   - `classical.FunExt`
   - `classical.Choice`
5. See the appendix for a detailed description of the technical development and instructions for building the mechanization manually.

4 Tested platforms

Virtualbox 7.2.4 running with 6 GB of RAM, 4 CPUs and 16 GB disk on Ubuntu 22.04.

5 License

The artifact is available under MIT license.

6 MD5 sum of the artifact

6e28f3df96526c9b5ca22f296fac4a4c

7 Size of the artifact

6.1 GB

A Correspondence between the paper and Rocq mechanization

A.1 Section 2

- General definitions and lemmas about traces: `trillium/traces/*.v`

A.1.1 Section 2.1

- Operational semantics of our language (Figure 3): `lawyer/heap_lang/lang.v`
- Calls and returns (Definition 1): `lawyer/nonblocking/trace_context.v`, definitions `call_at` and `return_at`. Note that the former explictly mentions the call argument, whereas the definition in the paper existentially quantifies over it.
- Eventual return of calls (Definition 2): `lawyer/nonblocking/wfree_traces.v`, definition `always_returns_strong`. Note that it is additionally parameterized with:
  - stuckness bit (thus covering the possibly-stuck definition)
  - predicate on the call argument. **This parameter is always set to an always true predicate and thus can be ignored.**
- Call fairness ("schedUntilRet"): `lawyer/nonblocking/wfree_traces.v`, definition `fair_call_strong`.
- Client validity: `lawyer/nonblocking/logrel/valid_client.v`, definition `valid_client`.
- Wait-freedom (Definition 3): `lawyer/nonblocking/wfree_traces.v`, definition `wait_free_strong`. Again, it is parameterized by a stuckness bit and an unused predicate on call arguments.

A.1.2 Section 2.2

- Lawyer specification of wait-freedom (Definition 4): `lawyer/nonblocking/om_wfree_inst.v`, record `WaitFreeSpec`. Note the following:
  - This specification explicitly mentions an invariant that should be established from the starting configuration and which is assumed by both Hoare triples. In contrast, Definition 4 does not mention a module invariant explicitly and rather allows to take a viewshift from starting configuration before proving the Hoare triples. However, viewshifts allow establishing invariants, and the proofs in the paper (Sec. 4) proceed exactly by establishing a module invariant. See Iris Lecture Notes, Sec 4.3 "Abstract Data Types" for the discussion on these specification styles.

  - Parameter `P` can be ignored
  - The amount of fuel consumed by the operation is determined by the *fuel function* `wfs_F`, mentioned in Sec. 5.1
  - We prohibit the wait-free operation from forking using the *forking bit*. For that, we use our variation of Trillium weakest precondition defined in `trillium/bi/weakestpre.v`.
- Wait-freedom adequacy theorem (Theorem 5): `lawyer/nonblocking/wfree_adequacy.v`, theorem `wfree_is_wait_free`. Again, it is parameterized by a stuckness bit, and the unused predicate on call arguments is set to be always true. It also explicitly requires the value representing the wait-free operation to be a lambda-expression.

A.2 Section 3

We do not mechanize the proofs presented in this section, as they are only used for explaining the Iris logic and not for establishing wait-freedom of `incr` (which is done in Section 4). The specifications and proofs are standard and explained in *e.g.*, Iris Lecture Notes [1].

A.3 Section 4

- Verification of the wait-freedom specification for the counter example:
  `lawyer/nonblocking/examples/counter/counter.v`, definition `counter_WF_spec`.
  Note that we use the two-step logic of Lawyer to verify the Lawyer triples. In this logic, verifying every step amounts to applying two rules: one for the physical execution step and one for the model step. The former rules (e.g. `wp_faa`) are listed in `lawyer/heap_lang/sswp_logic.v`, whereas the latter are implicitly applied by tactics such as `MU_by_burn_cp`.
- Wait-freedom of counter example:
  `lawyer/nonblocking/examples/counter/counter_adequacy.v`.
- The degree parameter of fuel is always set to the lowest degree `d0` of our Obligations Model instantiation: see `wfs_spec` in `WaitFreeSpec` located in
  `lawyer/nonblocking/om_wfree_inst.v`.
- The fraction parameter of phase is always set to `1/2`: see the Hoare triple in
  `wait_free_method_gen` located in `lawyer/nonblocking/om_wfree_inst.v`.
- `NoInfExec` Lawyer triple for wait-freedom: `lawyer/nonblocking/om_wfree_inst.v`, definition `wait_free_method_gen`. Ignore the `P` and `Q` parameters.
- `PresInv` triple for wait-freedom: defined directly as value interpretation (see Sec. 6.2)

A.4 Section 5

A.4.1 Section 5.1

- Stuckness variations of definitions related to wait-freedom (Definitions 6 and those mentioned below it): they are specific cases of definitions used for Section 2.1 with stuckness bit set to `MaybeStuck`.
- Adequacy theorem for possibly-stuck wait-freedom:
  instantiation of `lawyer/nonblocking/wfree_adequacy.v`,
  theorem `wfree_is_wait_free` with stuckness bit set to `MaybeStuck`.
- Modular verification of `list_map`: `lawyer/nonblocking/examples/list_map/list_map.v`
  - `list_map` implementation (Figure 6a): `hl_list_map_cur`.
  - Modular proof of specification: `hlm_WF_fix_spec_unsafe`. Note that we verify wait-freedom for eta-expanded `hl_list_map_cur f` due to the lambda-expression restriction (mentioned above).

- Possibly-stuck wait-fredom of `list_map (incr l)`:
  `lawyer/nonblocking/examples/list_map/list_map_adequacy.v`.
- Fuel function: `wfs_F` component of `WaitFreeSpec`.

A.4.2 Section 5.2

Note that throughout the restricted wait-freedom development we use multisets of operations instead of lists.

- Implementation of the queue algorithm (Figure 7) in our language is scattered across multiple files in `lawyer/nonblocking/examples/queue`:
  - `dequeuer/dequeue.v`, definition `dequeue`
  - `dequeuer/read_head_dequeuer.v`, definition `read_head_dequeuer`
  - `dequeuer/dequeuer_thread.v`, definition `dequeuer_thread`
  - `enqueuer/enqueue.v`, definition `enqueue`
  - `enqueuer/read_head.v`, definition `read_head_enqueuer`
  - `enqueuer/enqueuer_thread.v`, definition `enqueuer_thread`
- Restricted wait-freedom (Definition 7): `lawyer/nonblocking/wfree_traces.v`, definition `wait_free_restr`.
- Exclusion of forks: `lawyer/nonblocking/logrel/valid_client.v`, definition `no_forks`.
- Specification of restricted wait-freedom (Definition 8):
  `lawyer/nonblocking/tokens/om_wfree_inst_tokens.v`, definition `WaitFreeSpecToken`.
- Definition and lemmas about tokens resource algebra:
  `lawyer/nonblocking/tokens/tokens_ra.v`
- Adequacy theorem for restricted wait-freedom (Theorem 9):
  `lawyer/nonblocking/tokens/wfree_adequacy_tokens.v`,
  theorem `wfree_token_is_wait_free_restr`.
- Restricted wait-freedom of the queue algorithm:
  `lawyer/nonblocking/examples/queue/simple_queue_adequacy.v`

A.5 Section 6

A.5.1 Section 6.1

- Reduction to proving termination: it is scattered across multiple lemmas used to prove `wfree_is_wait_free` mentioned above. In particular, see the lemmas in `WFAdequacy` section which fixes the parameters of an infinite call.
- Definition of progress resource: `trillium/trillium/program_logic/adequacy_cond.v`, record `ProgressResource`. Note that it is additionally parameterized with stuckness and forking bits (and list of postconditions mentioned in the appendix).
- Conditional adequacy theorem of Trillium (Theorem 10):
  `trillium/trillium/program_logic/simulation_adequacy_em_cond.v`,
  theorem `PR_strong_simulation_adequacy_traces_multiple`.
- Refinement relation for wait-freedom: `lawyer/nonblocking/wfree_adequacy_lib.v`, definition `obls_sim_rel_wfree`

A.5.2 Section 6.2

- Expression relation: `lawyer/nonblocking/logrel/logrel.v`, definition `interp_expr`
- Value relation (Definition 11): `lawyer/nonblocking/logrel/logrel.v`, definition `interp`
- Fundamental theorem (Theorem 12): `lawyer/nonblocking/logrel/fundamental.v`, theorem `fundamental`
- Robust safety of the wait-free operation (Theorem 13): `lawyer/nonblocking/wfree_adequacy.v`, definition `init_wptp_wfree`

A.5.3 Section 6.3

- Definition of progress resource for wait-fredom: `lawyer/lawyer/nonblocking/pr_wfree.v`, definition `pr_pr_wfree`.
- Definition of the `infCallPrefix` predicate: `lawyer/nonblocking/wfree_traces.v`, definition `fits_inf_call`.
- Proof of the progress resource laws: `lawyer/lawyer/nonblocking/pr_wfree.v`, definition `PR_wfree`.

A.6 Extra

- "Wait-freedom" of a simple sequential program: `lawyer/nonblocking/examples/mk_ref/(mk_ref, mk_ref_adequacy).v`
- Variations of above definitions and theorems for restricted wait-freedom: `lawyer/nonblocking/tokens/*.v`. In particular:
  - Extension of trace intepretation for physical WP that keeps track of method tokens: `pwp_ext.v`
  - Logical relation and fundamental theorem for token-based specifications: `logrel_tok.v` and `fundamental_tok.v`
  - Lifting of a stronger specification to one required by token-based FTLR: `op_spec_lifting.v`, lemma `lift_spec`
  - Progress resource for restricted wait-freedom: `pr_wfree_tokens.v`

B Building the mechanization manually

First, install `opam` according to the instructions for your operating system. We used the version 2.1.2. Then execute the following:

```
# unpack the files (assuming the supplementary material is in wfree_suppl.zip)
unzip -d wfree_suppl wfree_suppl.zip
# move into the working directory
cd wfree_suppl

# create a new opam environment
opam switch create wfree-env 5.2.0
# switch into the new environment
eval $(opam env --switch=wfree-env)

# set up repository for Rocq packages
opam repo add coq-released https://coq.inria.fr/opam/released
# set up the local repository for Trillium
opam pin add trillium trillium/ --no-action
```

```
# move into the Lawyer directory
cd lawyer/
# install all dependencies of Lawyer
opam install . --deps-only
# build Lawyer; adjust the number of jobs as needed
make -j 5
```

References

1 Lars Birkedal and Ales Bizjak. Lecture notes on Iris: Higher-order concurrent separation logic, 2017. URL: http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf.

2 Egor Namakonov, Lars Birkedal, and Amin Timany. Verifying wait-freedom for concurrent higher-order programs. *40th European Conference on Object-Oriented Programming (ECOOP 2026)*, 372, June 2026. doi:10.4230/LIPIcs.ECOOP.2026.20.

3 Egor Namakonov, Justus Fasse, Bart Jacobs, Lars Birkedal, and Amin Timany. Lawyer: Modular obligations-based liveness reasoning in higher-order impredicative concurrent separation logic. *Proc. ACM Program. Lang.*, 10(OOPSLA1), 2026. doi:10.1145/3798240.