Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris

Ike Mulder
Radboud University Nijmegen
The Netherlands
me@ikemulder.nl

Robbert Krebbers
Radboud University Nijmegen
The Netherlands
mail@robbertkrebbers.nl

Herman Geuvers
Radboud University Nijmegen and Eindhoven University of Technology
The Netherlands
herman@cs.ru.nl

Abstract
Fine-grained concurrent programs are difficult to get right, yet play an important role in modern-day computers. We want to prove strong specifications of such programs, with minimal user effort, in a trustworthy way. In this paper, we present Diaframe—an automated and foundational verification tool for fine-grained concurrent programs.

Diaframe is built on top of the Iris framework for higher-order concurrent separation logic in Coq, which already has a foundational soundness proof and the ability to give strong specifications, but lacks automation. Diaframe equips Iris with strong automation using a novel, extendable, goal-directed proof search strategy, using ideas from linear logic programming and bi-abduction. A benchmark of 24 examples from the literature shows that the proof burden of Diaframe is competitive with existing non-foundational tools, while its expressivity and soundness guarantees are stronger.

CCS Concepts:
• Theory of computation → Separation logic; Automated reasoning; Program verification.

Keywords: Separation logic, fine-grained concurrency, proof automation, Iris, Coq

ACM Reference Format:

1 Introduction
Fine-grained concurrent programs, such as locks, reference counters, barriers, and queues, play a critical role in modern day programs and operating systems. Based on 15 years of research on concurrent separation logic [12, 13, 25, 29, 30, 32–35, 48, 67, 68, 74, 80, 81, 85–89], it has become possible to verify increasingly complicated versions of such programs. Yet, while several tools for verification of fine-grained concurrent programs based on these logics exist, none of them are both automated (the majority of the proof work is carried out by the tool) and foundational (a closed proof w.r.t. the operational semantics is produced in a proof assistant).

Tools with good automation like Caper [31], Starling [90] and Voila [91], generally use SMT [27] or separation-logic solvers [65, 73] as trusted oracles. They are capable of proving programs correct with relatively little help from the user, allowing quick experimentation when designing algorithms. However, they have a large trusted computing base—one needs to trust their implementation, the used solvers, the translation of the required side conditions to the used solvers, and sometimes also the soundness of the underpinned logic. In particular, the results of such tools do not come with closed proofs that can be checked independently.

Foundational tools like Iris [45, 46, 48, 52], FCSL [77] and VST [3, 17] are embedded in a proof assistant. Hence, one only needs to trust the implementation of the proof assistant and the operational semantics of the programming language, but not the solvers or underpinned logic. Foundational tools typically provide tactics [2, 6, 17, 51, 53, 60] to hide low-level proofs, but the bulk of the proof work needs to be spelled out. There are two reasons for this status quo. First, foundational tools cannot rely on trusted oracles, unless proofs are reconstructed so that the proof assistant can verify them independently. Second, foundational tools usually have a rich logic that can prove strong specifications, e.g., using impredicative invariants [80], for which automation has received little attention, even in a non-foundational setting.

In this paper, we present Diaframe—a foundational tool for automatic verification of fine-grained concurrent programs. Diaframe extends Iris [45, 46, 48, 52]—a framework for interactive proofs in higher-order impredicative concurrent separation logic in Coq—with powerful tactics to perform the bulk of the proof work automatically. This means we get the best of both worlds: closed proofs to underpin our results, while needing relatively little help from the user.

An overview of the architecture of Diaframe is displayed in Figure 1. Diaframe takes two inputs from the user (marked...
in blue)—a program with a Hoare-style specification, and optionally a set of user-provided hints. The program and specification are turned into an Iris entailment that we prove using an extendable, goal-directed proof search strategy. Inspired by seminal work on linear logic programming [43] and recent work on separation logic programming [76], our strategy interprets logical connectives as proof search instructions. These instructions simplify and solve (a part of) this entailment, possibly generating remaining proof obligations in the process. To make progress on the remaining obligations, our strategy looks for applicable hints.

Identifying good hints is one of the main challenges that we face. The proof rules of expressive logics like Iris (in particular, rules for invariants and ghost state) are not syntax directed and therefore hard to apply automatically. We identify a suitable hint and entailment format that makes it possible to mechanically find and instantiate the appropriate hints. Iris’s rules for symbolic execution, reasoning with invariants, and ghost state are translated into syntax-directed variants that match the hint format. An important feature of our entailment and hint format is that it supports a sufficiently large set of Iris’s proof rules, while at the same time allowing for an efficient implementation with little backtracking. We achieve this by taking inspiration from bi-abduction [15], but adding novel ideas to support Iris’s modalities and to postpone instantiation of existentials, which are both needed to support Iris’s invariant and ghost state mechanism.

Due to Iris’s expressive logic, which includes higher-order quantification, impredicative invariants, and the entirety of Coq’s logic, our proof strategy is inherently incomplete. Nonetheless, it is able to completely solve many verification problems in practice. We achieve this by letting our proof strategy (and entailment and hint format) focus on a subset of expressible Iris goals that often appear in formal verification. The proof strategy makes good partial progress on remaining goals, where it allows the user to help out with an interactive proof or custom proof hints.

2 Diaframe by Example

In this section we showcase Diaframe by verifying a spin lock (§2.1) and an Atomic Reference Counter (ARC) (§2.2). For both examples we will give Hoare-style specifications \{P\} e \{Φ\} in Iris, where P : iProp is a separation logic assertion and Φ : Val → iProp a separation logic predicate on values. The triple \{P\} e \{Φ\} means that for each thread that owns resources satisfying P, executing e is safe, and if the execution terminates with value w, the thread will end up owning resources satisfying Φ w. The dependency on w allows us to give expected return values in specifications. Note that Iris uses partial, not total correctness. We use the notation SPEC \{P\} e \{Φ\} for \{P\} e \{w. ∃®. ø w = w^3 + Q\} to more succinctly specify return values. We are explicit about the embedding ↯Φ ↯ of pure Coq proposition φ into Iris.

2.1 Verification of a Spinlock

Lines 1–8 in Figure 2 give the implementation of a spin lock in Iris’s default ML-like language HeapLang [45]. The newlock method creates a new lock in the unlocked state by allocating a new location with value false. The acquire method uses Compare And Set (CAS) to atomically compare the stored value of 1 to false, and only if these are equal, set it to true. It returns a Boolean to indicate if the equality test was successful. If the CAS succeeds, we have acquired the lock. If it fails, we spin by recursively calling the acquire method. To release the lock, the release method puts the lock back to the unlocked state (false).

Let us now consider the specification of the lock methods, given in lines 15–26 in Figure 2. These specifications use the representation predicates is_lock γ lk R and locked γ for locks [41, 80]. Here, is_lock γ lk R expresses that the lock at location lk protects assertions R, and locked γ expresses that the lock is in locked state. The ghost identifier γ is used to tie these two representation predicates together.

Given an arbitrary assertion R, the newlock method returns a value lk, for which is_lock γ lk R holds. The assertion is_lock γ lk R is duplicable, meaning it can be shared...
freely with multiple threads, and thus allows for multiple threads to call acquire in parallel. Calling acquire on a lock will result in evidence locked γ that the lock is locked, and access to assertion R. Contrary to is_lock γ lk R, the assertion locked γ is not duplicable, because at most one thread can hold the lock. To call release, we need to relinquish both locked γ and R, and get nothing in return.

Specifications of concurrent data structures based on representation predicates [30] allow for easy verification of clients by abstracting away from the implementation. The is_lock γ lk R representation predicate is particularly flexible, since it is impredicative [80]—meaning that the resources protected by the lock are described by an arbitrary separation logic predicate R that can contain other locks, Hoare triples, etc. To define impredicative representation predicates, we use Iris’s invariant and ghost state mechanism.

Programs using fine-grained concurrency have multiple threads reading and mutating shared state. In the example, the location backing the spinlock needs to be shared so that multiple threads can attempt to acquire the lock in parallel. Since the points-to assertion ℓ ↦ v of separation logic expresses exclusive ownership of the location ℓ with value v, we cannot just share it between multiple threads.

To reason about shared mutable state, we use Iris’s invariant assertion \( L^N \), which says that there is a (shared) invariant with name \( N \) governing the resources satisfying Iris assertion \( L \). Invariants \( L^N \) are duplicable, which means that the assertion \( L \) inside the invariant is accessible by all threads. To do this soundly, access to \( L \) is restricted. Only during atomic operations (like an assignment or CAS), invariants may be ‘opened’, which gives one temporary access to the assertion \( L \) in the verification of a thread. After the atomic operation, the invariant must be ‘closed’, meaning one must show the assertion \( L \) still holds.

Lines 9–14 contain the definition of is_lock γ lk R. It says that a value ik is a lock if it is equal to some location l, whose stored value is governed by an invariant lock_inv. Note that in Coq, we write inv N L for \( L^N \). The invariant lock_inv states that 1 should point to a Boolean. If this Boolean is true, the lock is locked, and we know nothing else since the resources satisfying \( R \) are currently owned by a thread which acquired the lock. If this Boolean is false, the lock is unlocked, and the resources satisfying \( R \) as well as the locked γ assertion are owned by the invariant.

The key ingredient for the verification of the spinlock is the ghost assertion locked γ, whose rules are:

\[
\text{locked-allocate} \quad \text{locked-unique} \\
\vdash \exists y. \text{locked } y \quad \text{locked } y \cdot \text{locked } y + \text{False}
\]

The first rule is used in the proof of newlock. It allows for the allocation of locked γ with a fresh ghost name y. This assertion is needed to establish the invariant by proving the right disjunct of lock_inv. (The update modality \( \exists y \) signifies a logical update to the ghost state. It will be explained in §3.2, but for now, it is enough to know that after each program statement, we can perform a logical update in the proof.)

The second rule states that locked γ is a singleton—no two threads/resources can simultaneously satisfy this assertion. This means that the locked γ assertion gives us information about the global state. In the proof of release, just before executing the store, the right disjunct of lock_inv is contradictory because locked γ is in the precondition. Hence, the left disjunct must hold—the location 1 must point to the value true, i.e., the lock is in locked state.

The general structure of verification in Diaframe is similar for other examples: we give the implementation and specification, and an invariant using appropriate ghost assertions, after which the verification will go mostly automatically. Other concurrent programs may use different ghost assertions, but all of these assertions have three types of rules: (a) allocation/creation rules, like locked_allocate, (b) compatibility/interaction rules, like locked_unique, and (c) mutation/update rules, of the form \( P + Q \vdash R * S \). We will see some update rules in the next example.

---

1For readers familiar with Iris, we simply define locked γ := \( L^N \).
We will now verify a version of an Atomic Reference Counter (ARC), similar to the one verified by Starling [90] and the one used in the Rust standard library [54]. An ARC can be used to safely give multiple threads read-access to a resource, while being able to recover write-access once all read-access references have been dropped. Lines 2–13 in Figure 3 give the implementation. Values of ARC are locations that store an integer containing the number of read-access references. The mk_arc method allocates a location with value 1, i.e., an ARC with one read-access reference. The count method gives the number of read-access references. The clone method increments the reference count with 1, using the atomic Fetch And Add (FAA) instruction, while drop decrements the reference count with 1. The unwrap method is like drop in that it will decrement the reference count—but by using a CAS operation to set the reference count from 1 to 0, it ensures that it destroys the last reference, and spins as long as other references have not been dropped.

To give a specification of the methods of ARC, we make use of shareable assertions, which are typically modeled with fractional permissions [11]. In Iris, shareable assertions are modeled as Iris predicates \( P : Q_p \rightarrow \text{iProp} \), where \( \text{iProp} \) is the type of Iris assertions, and \( Q_p \equiv \{ q \in Q \mid q > 0 \} \). Predicates \( P \) of this type must satisfy \( P q_1 \vdash P q_2 \vdash P (q_1 + q_2) \) to be called shareable (or Fractional in Coq). An example of a shareable assertion is the fractional mapsto connective \( \ell \mapsto q \). If \( q = 1 \), it denotes full ownership of (or write-access to) heap-location \( \ell \). If \( 0 < q < 1 \), it denotes fractional ownership of (or read-access to) heap-location \( \ell \).

As shown on line 1 in Figure 3, the whole verification is abstracted over a shareable assertion \( P \) that describes the resources that are being protected by the ARC. The specification of the methods can be found in lines 20–43. Like for the spinlock, we use several representation predicates. The duplicate assertion \( \text{is}_\text{arc} \triangleright \gamma \triangleright \nu \) says that a value \( \nu \) is an ARC. The non-duplicate assertion \( \text{token} \triangleright \gamma \triangleright \nu \) indicates that write-access to \( \nu \) exists. The non-duplicate assertion \( \text{no\_tokens} \triangleright \gamma \triangleright \nu \) indicates that write-access has been recovered, i.e., that no read-access tokens \( \nu \) exist.

With these predicates at hand, the specification of \( \text{mk}_\text{arc} \) requires \( P \) (write-access) and returns a value that \( \text{is}_\text{arc} \) guarding \( P \), along with a single read-access token. The count method is essentially a no-op, but shows that if we have a single-read access token, the reference count must be positive. The method \( \text{clone} \) duplicates a read-access token—it requires one of them, and returns two. The method \( \text{drop} \) destroys a token, and either returns nothing, or, if this was the last token, write-access \( P \), along with the knowledge that \( \text{no\_tokens} \) exist. The unwrap method, when it terminates, guarantees retrieving write-access \( P \) and \( \text{no\_tokens} \).

Let us look at the definition of \( \text{is}_\text{arc} \) in lines 14–19 in Figure 3. Similar to \( \text{locked} \), we treat token and \( \text{no\_tokens} \) abstractly (these are defined via Iris’s extensible ghost state mechanism, see our appendix [64] for the definition), and show only the allocation, interaction and update rules in Figure 4. As witnessed by \( \text{token\_access} \), these ghost-state assertions are used to convert fractional permissions into counting permissions [9], which are more natural for ARC.

Similar to the spinlock, we define a value to be \( \text{is}_\text{arc} \) if it is a location whose stored value is governed by an invariant.
This invariant `arc_inv` tells us that the location points to some integer `z`, which satisfies: (1) `z = 0`, and we know that no tokens currently exist, or (2) `z > 0`, and we own resources satisfying `counter P y z`. The counter `P y p` assertion states the knowledge that precisely `p` tokens currently exist—which matches what we want `ℓ = p` to mean.

To prove the specification of the count method, we use `TOKEN_ALLOCATE`, which allows us to establish the left disjunct of `arc_inv`. For proving the specification of count, we rely on `TOKEN_INTERACT` to prove that the right disjunct of `arc_inv` is contradictory. For the specification of clone, we again need `TOKEN_INTERACT`. When closing the invariant, we need to apply `TOKEN_MUTATE_INCR` at the right moment to change the obtained counter `P y p` to the required counter `P y (p + 1)`. This also gives us the extra token that we need in the postcondition.

**Integration with interactive proofs.** In the verification of drop, Diaframe encounters a goal it cannot solve automatically, and gets stuck. The user is presented with the following (slightly simplified) proof state in the Iris Proof Mode [51, 53], where they can use Coq or Iris tactics to help:

```coq
H : 0 < x2

"H1" : inv N (arc_inv y 1)
----------------------------------------□
"H2" : token P y

"H5" : counter P y (Z.to_pos x2)
----------------------------------------□

\( \bigl( \forall x1. N x1 \bigr) \) \( \text{mp} \) \( \{(N)\} \)

\( \{0 < x2 + \mathbf{1} \rightarrow \text{counter } P y (Z.to_pos (x2 + \mathbf{1}))\} \)
\( \lor \{x2 + \mathbf{1} = 0 \rightarrow \text{no_tokens } P y\} \)
\( \{\} \) \( \text{mp} \) \( \mathbf{1} \) \( \{\{\} \} \)
```

The statement below `→→` indicates our current goal, and contains a disjunction. Both sides of the disjunction contain a pure statement `Γ φ`, but neither of these follow from the relevant hypothesis `H`. On inspection, we need to distinguish two cases: `x2 = 1` and `x2 > 1`. In the first case, our token was the last one, and we need to use `TOKEN_MUTATE_DELETE_LAST` to finish the proof. In the second case, other tokens remain, and we need to use `TOKEN_MUTATE_DECAR`.

In Figure 3, the manual step consists precisely of the case distinction between `x2 = 1` and `x2 > 1`, after which Diaframe’s iSteps5 can finish the proof. Even though Diaframe could not figure out the required case distinction automatically, it makes good partial progress here. This is because the automation only performs limited backtracking, and simply stops when it encounters a goal it cannot make progress on.

**Generality.** The ghost assertions `token_no_token` and `counter` are not connected to a memory location and are thus not specific for the verification of ARC. We also use them in the verification of e.g., reader-writer locks. The only connection between these assertions and the ARC lies in the definition of the invariant `arc_inv`, which ties the physical state of the ARC to an appropriate ghost-state. The rules for the assertions in Figure 4 are available to the Diaframe proof search strategy, and applying them requires no extra annotations, except for the manual case distinction for drop.

### 3 Diaframe’s Entailment Format

In this section we explain some of the challenges one faces when automating proofs of fine-grained concurrent programs in Iris. We start with some background on verifying weakest preconditions of sequential programs using symbolic execution (§3.1), as commonly done in interactive and automatic tools in proof assistants [20, 53, 76]. We then extend this approach with support for Iris’s invariant mechanism to verify fine-grained concurrent programs (§3.2). We conclude with an overview of the Diaframe entailment format and proof strategy (§3.3), which serves as a starting point for the description of our hint format (§4).

#### 3.1 Goal-Directed Reasoning with WP

Hoare triples are not a primitive of Iris, they are defined in terms of weakest preconditions:

\[ (P) e (\{\Phi\}) \models P \vdash wp e (\{\Phi\}). \]

To get some intuition for the semantics of `wp e (\{\Phi\})`, assume for a moment that `P` and `Q` are predicates on heaps (ignoring Iris’s ghost state and step-indexing), and `Φ` is a predicate on values and heaps. Entailment `P \models Q` means that for every heap `h`, if `P h` holds, then `Q h` holds. The assertion `wp e (\{\Phi\})` describes the heaps for which execution of `e` is safe (cannot get stuck), and if `e` terminates with value `v` and heap `h`, then `Φ v h` holds. Defining \( (P) e (\{\Phi\}) \) as above then indeed gives the Hoare triple its intended and intuitive semantics.

Weakest preconditions make it possible to decouple the precondition from the Hoare triple, and view it as a regular
we need as part of the precon-

When we find an hypothesis of shape

this is provable, but do not yet know which one it will be.

we have put a question mark in front

rule

ramified frame

transform our entailment into

wp-faa

we need to find a 'frame'

wp-faa-ramify

Suppose we are proving the following entailment:

we can generalize the rule wp-faa-ramify to any Hoare-

style specification of an expression e:

This rule additionally incorporates Iris’s rule wp-bind, which

allows the expression e to appear inside a call-by-value evalua-

tion context K, instead of at the top-level.

Supposing we can prove separating conjunctions, sym-ex

gives rise to a symbolic-execution based proof search strategy

for straight-line sequential code. Suppose our goal is

Δ ⊢ wp e (Φ). If e is a value v, apply wp-value and prove

Δ ⊢ v. R + Φ v. Else, find an evaluation context K and subexpres-

sion e' with e = K[e'], and a specification {P} e' (Ψ). Apply

sym-ex, prove the separating conjunction, introduce vari-

ables, introduce the left-side of the magic wand, and repeat.

3.2 Goal-Directed Reasoning with Invariants

We will now extend the naive proof search strategy from

§3.1 with support for Iris’s invariant mechanism to handle

programs with fine-grained concurrency. Concretely, we will

present a rule that extends sym-ex, which can also be used

in case the precondition P is inside an invariant (as is the

case for all examples in §2). We will first recapitulate Iris’s

original proof rule for accessing invariants:

inv-open-wp

This rule is quite a mouthful, so let us go over it step by step.

First, to deal with invariants, weakest preconditions in Iris

wp_e e (Φ) have a mask annotation E, signifying the set of

names of invariants that can be opened. This is necessary

to ensure invariants are not opened more than once (i.e., to

avoid reentrancy, which is unsound). Omitted masks are T,

meaning all invariants can still be opened.

Suppose that we have an invariant \([\mathbf{I}]^N\), and are verifying

an atomic expression e. Rule inv-open-wp states that we are

allowed to look inside the invariant and obtain L in the proof

context, but then must show that L still holds in the postcon-

dition of the WP. After we have opened the invariant with

name N, the mask changes to E \(\setminus N\) so that we cannot open

the invariant twice. The later modality (\(\triangleright\)) \([4, 66]\) is needed

for technical reasons caused by the fact that invariants are

impredicative \([46, 80]\), i.e., the resource L in an invariant can be

any resource, including invariants and weakest precondi-

tions. Handling later modalities involves some additional

bookkeeping, which Diaframe performs automatically, but we

gloss over in this paper.

We now show why our sym-ex rule for symbolic execution

from §3.1 needs to be extended for programs involving

fine-grained concurrency. Consider the FAA operation in the

cClone method of the ARC (§2.2). The challenge of verifying

this method is that the \(\ell \mapsto \_\) we need as part of the precon-

dition for FAA is not in the proof context, but in an invariant

\(\exists (z : Z). \ell \mapsto z / \_\)

When we apply sym-ex eagerly, we lose the ability to open invariants using inv-open-wp.
One approach is to try to make progress with \textsc{sym-ex}—if this is possible, we are alright. If not, we backtrack, and open an invariant with \textsc{inv-open-wp}, and retry. This is similar to the approach employed by Caper [31]. We do not take a backtracking approach in Dioframe since it does not mix nicely with interactive proofs.

We therefore present an extended symbolic execution rule, \textsc{sym-ex}, which allows us to open invariants lazily:

\begin{align*}
\text{\textsc{sym-ex-fupd-exist}} & \\
\forall x. \{P\} e \{\Psi\} & \quad \text{atomic } e \lor \exists x = ?E2 \\
\Delta \vdash E1 \Rightarrow ?E2 & \exists x. P \times \left(\forall w. \Psi \wedge w \Rightarrow ?E2 \Rightarrow \begin{array}{l}
wp_{E1} K[w] \{\Phi\}
\end{array}\right) \\
\Delta \vdash \begin{array}{l}
wp_{E1} K[e] \{\Phi\}
\end{array}
\end{align*}

This rule contains Iris’s \textit{fancy update modality} $E1 \Rightarrow E2$, and a \textit{quantified Hoare triple} $\forall x. \{P\} e \{\Psi\}$.

The fancy update modality, $E1 \Rightarrow E2$, is used in Iris’s definition of weakest preconditions, and is the component that makes opening invariants possible. Semantically, $E1 \Rightarrow E2$ means: assuming all invariants with names in $E1$ hold, then $P$ holds and additionally all invariants with names in $E2$ hold.

To work with the fancy update invariant, Iris has the following rules:

\begin{align*}
\text{\textsc{inv-open-fupd}} & \\
N \subseteq E & \\
\begin{array}{l}
\Delta \vdash E \Rightarrow N \\
\vdash L \Rightarrow E \Rightarrow N \left(\begin{array}{l}
\Rightarrow L \Rightarrow E \Rightarrow N \Rightarrow \text{True}
\end{array}\right)
\end{array} & \\
P \vdash \begin{array}{l}
wp_{E1} K[e] \{\Phi\}
\end{array}
\end{align*}

The \textsc{inv-open-fupd} rule makes the semantics of invariants precise: by removing $N$ from the mask, we get access to $L$, and if we wish to restore the mask, we must hand back $L$ via the closing update ($\Rightarrow L \Rightarrow E \Rightarrow N \Rightarrow \text{True}$). The rule \textsc{fupd-elim} allows us to compose fancy updates, and by combining \textsc{bupd-fupd} and \textsc{bupd-intro} we can introduce the last fancy update when done. Note that \textsc{bupd-fupd} and \textsc{fupd-elim} enable us to perform logical updates (like those in Figure 4) when the goal contains a fancy update after the turnstile.

The quantified Hoare triple $\forall x. \{P\} e \{\Psi\}$ states that the Hoare triple $\{P\} e \{\Psi\}$ holds for all instantiations of the auxiliary variables in $x$. Here, $P$ should and $\Psi$ may refer to the variables in $x$. For FAA, we have:

\[\forall z1. \{\ell \mapsto z1\} \text{ FAA } \ell \vdash z1 \{w. \Rightarrow w = z1 \Rightarrow \ell \mapsto (z1 + z2)\} \]

The essential feature of \textsc{sym-ex-fupd-exist} is that once we apply the rule, we retain the ability to open (any number of) invariants through a combination of the rules \textsc{fupd-elim} and \textsc{inv-open-fupd}. Our new rule is strictly stronger than the rule \textsc{sym-ex} from §3.1—the update modalities can simply be introduced using \textsc{bupd-fupd} and \textsc{bupd-intro}, and the existentials can be instantiated with evars.

We now show why the existential quantification in the new rule is necessary. Let us try to use \textsc{sym-ex-fupd-exist} wrongly by instantiating existentials eagerly in a goal that arises during the verification of an FAA in ARC (§2.2):

\[
\begin{array}{l}
\ell \mapsto z1, \ldots \vdash ?E2 \Rightarrow \begin{array}{l}
wp_{E1} K[e] \{\Phi\}
\end{array}
\end{array}
\]

One should read this proof derivation from bottom to top. When encountering an FAA, we apply \textsc{sym-ex-fupd-exist}, but (wrongly) perform an eager instantiation of the existential $z'$ with an evar $?z1$. Then we use \textsc{inv-open-fupd} and \textsc{fupd-elim} to open the invariant. The final step uses some properties of the later modality to eliminate the existential and the later around $\ell \mapsto z$. One might think we are now done: just unify $?z1$ with $z$ and $?E2$ with $\top \Rightarrow \top$, and continue! However, this is not sound—the evar $?z1$ cannot be unified with $z$, since $z$ was introduced after $z1$. Stated in other words, we could not have chosen $z1$ to be equal to $z$, since at that point $z$ was not in our context. To correctly deal with existentials, the Dioframe proof search strategy delays the instantiation of existentials.

### 3.3 Overview of the Dioframe Strategy

To automatically prove program specifications $\forall x. \{P\} e \{\Psi\}$, Dioframe’s proof strategy repeatedly performs the following actions (a formal presentation is given in §5):

1. If the goal is $\Delta \vdash \forall x. G$ or $\Delta \vdash U \Rightarrow G$, introduce the $\forall$ or $\Rightarrow$. Then “clean” the hypothesis $U$ by (a) eliminating separating conjunctions, disjunctions, and existentials, (b) moving pure assertions $\Phi \Rightarrow \top$ into the Coq context, (c) merging assertions (e.g., $\ell \mapsto q \wedge \ell \mapsto_p v$ become $\ell \mapsto q \wedge v = w$, (d) deriving contradictions (e.g., using \textsc{locked-unique}).

2. If the goal is $\Delta \vdash \wp_{v} \{\Phi\}$, with $v$ a value, continue with $\Delta \vdash ?E \Rightarrow ?E \Rightarrow \top \Rightarrow \top$.

3. If the goal is $\Delta \vdash \wp_{v} K[e] \{\Phi\}$, use our new symbolic execution rule \textsc{sym-ex-fupd-exist}. Our new goal has the shape $\Delta \vdash E1 \Rightarrow E2 \Rightarrow \exists x. L \Rightarrow G$.

4. If the goal is $\Delta \vdash E1 \Rightarrow E2 \Rightarrow \exists x. L \Rightarrow G$, use associativity of separating conjunction to rewire it into $E1 \Rightarrow E2 \Rightarrow \exists x. L \Rightarrow G'$ where $A$ is an atom. Pure conditions $r \Rightarrow r \Rightarrow \top$ that appear in the process are solved with Coq tactics like \textsc{lia}. We make progress on $A$ by finding a hint.

For this strategy to be effective, finding hints (in the last step) is crucial. These hints need to make sure that the resulting goal is again of one of the above entailment formats so the strategy can make repeated progress. When operating on entailments of format $\Delta \vdash E1 \Rightarrow E2 \Rightarrow \exists x. L \Rightarrow G$, it is essential that modalities and existentials are only introduced/instantiated when the right invariants have been opened and the necessary ghost updates have been performed—not earlier.
The Diaframe proof strategy is inspired by the idea of interpreting logical connectives as instructions to control the proof search, as done in the seminal work on linear logic programming [19, 43] and recent work on the separation logic programming language Lithium [76]. Other recent work by Chlipala [21, 22] has also shown that using the syntax of the goal to guide proof search works well for automatic foundational verification. The inspiration by Lithium can be seen most clearly in the reversible actions described in Items (a) and (b)—these are the same as those performed by Lithium. The key difference is that we do not operate on top-level connectives, but on connectives that appear below a modality and a number of existentials, to support Iris’s invariants and ghost state.

4 Diaframe’s Hint Format

In this section, we describe the process of finding hints. We consider the following kinds of base hints: (a) hints for ghost state such as those corresponding to the rules in Figure 4, (b) hints for language-specific connectives such as the \( \Rightarrow \) connective, and (c) user-defined hints to guide the proof of a specific program in case the automation falls short.

There are two ways how hints can be selected. First, goal-and-hypothesis directed hints use the shape of the goal and the shape of a hypothesis as keys. Examples are hints for mutating ghost state. Second, last-resort goal-directed hints are used if no hints that key on a hypothesis can be found. Examples are invariant allocation and ghost state allocation.

Hints are specified using a hint format (§4.1) that is inspired by the technique of bi-abduction [15]. Aside from the base hints (§4.2), Diaframe provides recursive hints to close the base hints under connectives like invariants, magic wands, and separating conjunctions (§4.3).

4.1 Bi-Abduction Hints

The hint format of Diaframe is as follows:

\[
H \Rightarrow [\bar{y}; L] \equiv [\bar{e}_1] \Rightarrow E_2 \quad \bar{x}; A \equiv [U] \quad \forall \bar{y}. (H \Rightarrow \bar{e}_1 \Rightarrow E_2 \quad (\exists \bar{x}. A \equiv U))
\]

Hints use a hypothesis \( H \) and goal \( A \) as key/input. Outputs are denoted between \( [ ] \) syntax; \( L \) is a (possibly existentially quantified) side condition, while \( U \) is the residue we obtain after using the hint. The hypothesis \( H \) is \( \bar{e}_1 \) for a last-resort hint. The assertion \( \bar{e}_1 \) is an opaque marker whose semantics is True, but is treated differently by the proof search strategy.

It is instructive to check the scope of the existentials. The premise \( H \) is a given hypothesis, so \( \bar{x} \) and \( \bar{y} \) do not occur in \( H \). The conclusion \( A \) is a given existential goal, so \( \bar{x} \) occurs in \( A \), but \( \bar{y} \) does not. The side condition \( L \) is existentially quantified with \( \bar{y} \). The residue \( U \) is allowed to contain both \( \bar{x} \) and \( \bar{y} \) so it can be related to the side condition \( L \) and the goal \( A \).

We also call Diaframe’s hints “bi-abduction hints” because in essence, they are bi-abduction [15] behind a modality and existentials. The bi-abduction problem in separation logic asks to find, given an hypothesis \( H \) and goal \( A \), a ’frame’ and ’antiframe’ such that \( H \Rightarrow \text{antiframe} \Rightarrow A \Rightarrow \text{frame} \). Our hints’ shape is also similar to the residuation judgment from Cervesato et al. [19], but has an additional frame.

We can apply a Diaframe bi-abduction hint as follows:

\[
\text{biabd-hint-apply}
\]

\[
\Delta \vdash [\bar{e}_1] \Rightarrow E_2 \quad \exists \bar{y}. L \Rightarrow (\forall \bar{x}. U \Rightarrow G)
\]

\[
\Delta, H \vdash [\bar{e}_1] \Rightarrow [\bar{e}_2] \exists \bar{x}. A \equiv G
\]

The Diaframe implementation will go over the hypotheses \( H \) in the context \( \Delta \) from left to right (with \( \varepsilon \), last) until it finds a hint \( H \Rightarrow [\bar{y}; L] \Rightarrow [\bar{e}_1] \Rightarrow E_2 \quad \bar{x}; A \equiv [U] \) in the hint database. This involves some backtracking, but only locally—whenever a hint (and thus a side condition \( L \) and residue \( U \)) has been found for a hypothesis \( H \), we use that hint and will never backtrack to consider a different choice. Note that after applying the rule, the resulting entailment has the same format, allowing for repeated applications of hints.

4.2 Base Hints

**Example 1: Ghost state mutation.** We transform the rule `token-mutate-decr` (which is used to verify the drop method of ARC in §2.2) into the following hint:

\[
\text{counter } P \gamma p \Rightarrow [\_; \text{token } P \gamma p \Rightarrow r p > 1] \equiv [\bar{e}_1] \Rightarrow [\bar{e}_2] \quad \text{counter } P \gamma (p - 1) \Rightarrow [\text{True}]
\]

If we use `biabd-hint-apply` with this hint, we get:

\[
\Delta \vdash [\bar{e}_1] \Rightarrow [\bar{e}_2] \quad \text{token } P \gamma p \Rightarrow r p > 1 \Rightarrow (\text{True} \Rightarrow G)
\]

\[
\Delta, \text{counter } P \gamma p \vdash [\bar{e}_1] \Rightarrow [\bar{e}_2] \quad \text{counter } P \gamma (p - 1) \Rightarrow G
\]

Here we see that to decrement the counter, we need to solve the side condition token \( P \gamma, \) before we can continue with \( G \).

**Example 2: Invariant allocation.** In Iris, invariants are allocated using the rule `L \Rightarrow \bar{e}_1 \Rightarrow E_2 \Rightarrow L^N`, which we transform into the following hint:

\[
\varepsilon_1 \Rightarrow L \Rightarrow [\_; \text{L}^N] \Rightarrow [\bar{e}_1] \Rightarrow [\bar{e}_2] \quad [\_; \text{L}^N] \Rightarrow [\text{True}]
\]

Due to the \( \varepsilon_1 \), this is a last-resort goal-directed hint. We do not make it hypothesis directed, because \( \Rightarrow L \) will usually not be precisely in the context. Since invariants are duplicable we give back \( \text{L}^N \) in the residue, so that it can be used again.

**Example 3: Ghost state allocation.** We transform the rule `locked-allocate` (which is used to verify the newlock method in §2.1) into the following hint:

\[
\varepsilon_1 \Rightarrow [\_; \text{True}] \Rightarrow [\bar{e}_1] \Rightarrow [\bar{e}_2] \quad \text{locked } \gamma \Rightarrow [\text{True}]
\]

Due to the \( \varepsilon_1 \), this is again a last-resort goal-directed hint. That is simply because the rule has no premise.
**Example 4: Points-to assertion.** We have specific hints for HeapLang’s fractional points-to assertion $\ell \mapsto q \nu$:

$$\ell \mapsto q_1 \nu \mapsto \begin{bmatrix} \vdash \nu_1 = v_2 \end{bmatrix} \Rightarrow \begin{array}{l} \nu \mapsto q_2 \nu \mapsto \begin{bmatrix} \vdash \nu_1 = v_2 \end{barray} \end{array}$$

This hint says that if we have a points-to for $\ell$, but need one with another value, we should prove that both values are equal. The following hint handles different fractions:

$$\ell \mapsto q_1 \nu \mapsto \begin{bmatrix} \vdash \nu_1 = v_2 \end{barray} \Rightarrow \begin{array}{l} \nu \mapsto q_2 \nu \mapsto \begin{bmatrix} \vdash \nu_1 = v_2 \end{barray} \end{array}$$

This hint applies if the fraction $q_2$ in the goal is bigger than the fraction $q_1$ in the hypothesis, and has the side condition $\ell \mapsto q_1 \nu \mapsto \begin{bmatrix} \vdash \nu_1 = v_2 \end{barray}$. Note that $v_3$ is existentially quantified, meaning that the side condition can be established for any value. This is sound by the agreement property of $\mapsto$. This generality is used in the verification of e.g., the CLH-lock. There is a dual hint for the case $q_1 > q_2$.

### 4.3 Recursive hints

It is often the case that a base hint almost—but not precisely—matches. The premise might appear under a magic wand or in an invariant, or the goal might provide a specific witness while looking for an existential. Diaframe therefore includes a number of recursive hints to close the base hints under the connectives of higher-order separation logic. For example:

$$U = \vdash [Z, L_2] \Rightarrow [E_1] \Rightarrow \begin{array}{l} \nu \mapsto \begin{bmatrix} \vdash \nu_1 = v_2 \end{barray} \Rightarrow \begin{array}{l} \nu \mapsto \begin{bmatrix} \vdash \nu_1 = v_2 \end{barray} \end{array} \end{array}$$

This rule states that if there is a hint from the conclusion $U$ of the wand to the goal $A$, then there is a hint from the wand $L_1 \mapsto U$ itself, where the premise $L_1$ of the wand is added to the side condition $L_2$. A more complicated recursive hint is the rule for invariants:

$$\begin{array}{l} \nu \mapsto \begin{bmatrix} \vdash \nu_1 = v_2 \end{barray} \Rightarrow \begin{array}{l} \nu \mapsto \begin{bmatrix} \vdash \nu_1 = v_2 \end{barray} \end{array} \end{array}$$

This rule states that there is a hint from an invariant $[L_1] = \vdash Z \vdash \nu \subseteq E$ to a goal $A$, if there is a hint from the contained assertion $L_1$ to that atom. We get $N \subseteq E$ as an additional side condition, and receive the closing update ($\nu L_1 \mapsto E[N] = E[\chi]$) as the residue. Similar to $\nu_1$, the assertion $\chi$ is an opaque marker whose semantics is True, but is treated differently by the proof search strategy to enforce closing invariants.

### 5 Formal Description of the Proof Strategy

In this section we will present an excerpt of the formal grammar of Diaframe (§5.1), and a number of cases of the formal proof search strategy (§5.2). We then present an extension of Diaframe to handle disjunctions (§5.3).

#### 5.1 Grammar of Diaframe

We provide a representative subset of the grammar (a full description can be found in the appendix [64]):

- **Atoms** $A ::= \text{wp } e \{v. L\} | \chi | \begin{bmatrix} \nu \end{bmatrix} E | \ldots$
- **Left-goals** $L ::= \nu \gamma | A | L \bowtie L | \exists x. L$
- **Unstructured** $U ::= \nu \phi | A | U \bowtie U | \exists x. L$
- **Goals** $G ::= \forall x. G | U \mapsto G | \text{wp } e \{v. L\}$
- **Extended** $H ::= \epsilon | U$

The entailments we wish to solve are of the form $\Gamma; \Delta \vdash G$. The atoms $A$ by default only consist of weakest preconditions $\text{wp } e \{v. L\}$, the marker $\chi$ (§4.3) and invariants $[\nu E]$. The ellipsis (…) indicates that the set of atoms may be extended by libraries, adding language-specific constructs like $\ell \mapsto v$ or ghost assertions like $\text{lock } y$. The definition of $\Delta$ explicitly sets the last-resort marker $\epsilon_1$ as the last hypothesis. Defining $\Delta$ in this way avoids having special cases in the description of the strategy, and is close to the Coq implementation.

We have two syntactical categories related to hypotheses: $H_C$ and $U$. Essentially, $U$ is the class of hypotheses for which we are able to recursively find hints. At introduction into the context $\Delta$, we can decompose these into $H_C$. The goal $[\nu E] \bowtie E | \exists x. L \cdot R \cdot G$ in $G$ is a ‘synthetic’ representation of $[\nu E] \bowtie E | \exists x. L \cdot R \cdot G$. This condition ensures that during hint search we only consider the relevant variables for $L$. To uphold this condition, our strategy first transforms goals like $[\nu E] \bowtie E | \exists x. L \cdot R \cdot G$ into $\nu E \cdot \exists x. L \cdot R \mapsto G$.

#### 5.2 The Proof Search Strategy

If our goal is $G; \Delta \vdash G$, we do a case analysis on $G$:

1. $G = \forall x. G'$: Continue with $\Gamma, x; \Delta \vdash G'$.
2. $G = U \mapsto G'$: Case analysis on $U$:
   - $U = \nu \gamma$: Continue with $\Gamma, \nu; \Delta \vdash G'$.
   - $U = (U_1 \bowtie U_2)$: Continue with $\Gamma, \Delta \vdash U_1 \mapsto U_2 \mapsto G'$.
   - $U = \exists x. L$: Continue with $\Gamma; \Delta \vdash \forall x. (L \mapsto G')$.
   - $U = H_C$: Continue with $\Gamma; \Delta, H_C \vdash G'$.
3. $G = \text{wp } e \{v. L\}$:
   - If $e$ is a value $w$, continue with $\Gamma; \Delta \vdash \nu \gamma L[w/v]$.
   - Else, find a $K$ and $e'$ with $e = K[e']$, and quantified specification $\forall x. L$.
4. $G = \nu E \cdot \exists x. L$: We consider the following cases:
a. If the modality $E_1 \triangleright E_2$ is not introducible, continue with $\Gamma \vdash E_1 \triangleright E_2 \land \exists x. \phi \vee E_2$. The remaining cases assume that $E_1 \triangleright E_2$ is introducible.

b. $L = \neg \phi$: Prove the pure goal $\phi$ to finish.

c. $L = wp e \{v. L'\}$: Continue with $\Gamma; \Delta \vdash wp e \{v. L'\}$.

d. Continue with $\Gamma \vdash E_1 \triangleright E_2 \land \exists x. \phi \vee E_2 \land L \not\triangleright E_2$. The recursive algorithm [78] extensively to make the implementation para-

d. $L = \neg \phi$: Move the pure goal $\phi$ to finish.

c. $L = wp e \{v. L'\}$: Continue with $\Gamma; \Delta \vdash wp e \{v. L'\}$.

d. Continue with $\Gamma \vdash E_1 \triangleright E_2 \land \exists x. \phi \lor E_2 \land L \not\triangleright E_2$. True in all other cases.

5. $G = \{ E_1 \vdash E_2 \} \triangleright [\exists x. L \land G']$: Case analysis on $L$:

a. $L = \neg \phi$: Check that $E_1 \vdash E_2$ is introducible, and try to solve $\phi$. Continue with $\Gamma; \Delta \vdash G'$.

b. $L = L_1 \lor L_2$: Set $\bar{y}_1 = \text{FV}(L_1)$ and $\bar{y}_2 = \bar{y} \setminus \bar{y}_1$, continue with $\Gamma; \Delta \vdash E_1 \triangleright E_2 \land [\exists \bar{y}_1. L_1 \land E_2 \lor [\exists \bar{y}_2. L_2 \land G]$.

c. $L = \exists y. L'$: Continue with $\Gamma; \Delta \vdash E_1 \triangleright E_2 \land [\exists y. L' \lor G]$.

d. $L = \exists y. \exists \bar{x} L'$: Find the first $H \in \Delta$ with $L'$ and $U$ for which $H \vdash \exists y. L' \lor [\exists E_1 \triangleright E_2 \land \exists \bar{x} L' \lor (\forall \bar{x}. U \land \neg G)]$.

3. Extending Diaframe with Disjunctions

The Diaframe grammar does not contain disjunctions. This is intended, as proving disjunctions in linear logics is challenging. Consider $P \lor Q \vdash (P \lor Q) \lor P$. It is crucial to prove the disjunction using $Q$, since otherwise we are left with the unprovable goal $Q \lor P$. But if we look at just the disjunction, there is no way to know this in advance.

5.3 Extending Diaframe with Disjunctions

The Diaframe grammar does not contain disjunctions. This is intended, as proving disjunctions in linear logics is challenging. Consider $P \lor Q \vdash (P \lor Q) \lor P$. It is crucial to prove the disjunction using $Q$, since otherwise we are left with the unprovable goal $Q \lor P$. But if we look at just the disjunction, there is no way to know this in advance.

To prove automation for some goals with disjunctions, we provide an extension of Diaframe. When introducing a disjunction $\Gamma \vdash U_1 \lor U_2 \lor G$ into the context, continue with $\Delta \vdash U_1 \lor G$ and $\Delta \vdash U_2 \lor G$ by disjunction elimination. When proving $\Gamma; \Delta \vdash E_1 \triangleright E_2 \lor [\exists x. L \lor L_2 \lor G$ (and symmetrically), check if we can prove $\neg \phi$, and if so, continue with the simpler goal $\Gamma; \Delta \vdash E_1 \triangleright E_2 \lor [\exists x. L \lor L_2 \lor G$. This makes the pure goal $\phi$ act as a "guard" on the disjunct.

When a disjunction cannot be handled this way, the proof search strategy will simply stop. It is then up to the user to choose a disjunct, and continue the proof (see the proof of drop in §2.2 for an example). To automatically prove more involved examples, Diaframe allows users to opt-in on the use of backtracking to choose a disjunct.

6 Implementation and Evaluation

Diaframe is implemented as a library of ca. 15,000 lines of Coq code, built on top of Iris. We use Coq’s type class mech-

anism [78] extensively to make the implementation parametric in (among others) the base proof hints. The recursive hint search strategy (§4.3) and the core proof search strategy (§5.2) are implemented as an Ltac [28] tactic called rStepsS. This tactic can be used to prove specifications entirely, and as part of interactive proofs in the Iris Proof Mode [51, 53]. Diaframe comes equipped with 5 ghost-state libraries with bi-abduction hints, to help verify concurrent programs.

To evaluate Diaframe and its implementation, we have verified 24 examples with different levels of complexity. These examples include all the examples used to evaluate Capera [31], Starling [90] and Voila [91], and 5 additional, closely related examples. Our examples do not always correspond line-for-line to the examples from other tools, since the programming languages are different, but the required concurrency reasoning is similar. These examples and their statistics are shown in Figure 6. This table also includes statistics for manual Iris proofs (if they are available in Iris’s Coq distribution).

From this benchmark, we conclude that the use ofDiaframe significantly reduces the proof work when using Iris to formally verify programs. Diaframe is competitive with automatic non-foundational tools such as Starling, Voila and Capera, while being foundational—generating closed proofs in the Coq proof assistant. The following caveats apply: (a) Starlings constraint-based approach reduces the proof work for some examples, e.g., Peterson’s algorithm. For most examples, Diaframe requires less proof work, and is more expressive. (b) Capera outperforms Diaframe with respect to proof work and number of annotations. However, verification with Diaframe is modular, meaning it is easier to verify clients. (c) Voila focuses on TaDA-style logically-atomic specifications [25], which are not supported by Diaframe. Because of this focus, Voila requires more proof work than Diaframe, also for the regular specifications used in this comparison.

We summarize some aggregated data from Figure 6. Diaframe can verify 7 of the examples without any help from the user. Averaged over all examples, we require about 0.4 line of manual proof per line of implementation (521 lines of proof for 823 lines of implementation). The highest proof work is in the verification of the Michael-Scott queue [63], requiring 46 lines of proof for an implementation of 37 lines. All but two examples can be verified in under two minutes on our 3960X Threadripper (averaged over 10 runs). The two exceptions are slow mainly because their invariants contain an $n$-fold disjunction, with $n$ relatively high ($\geq 10$).

Hints and proof search customization. Diaframe has access to 30 bi-abduction hints, available in our 5 ghost-state libraries. In Figure 6, user-provided hints and their required lemmas count as proof search customization. 8 user-provided bi-abduction hints were necessary to verify examples with recursive definitions, as Diaframe does not have native support for such definitions as of yet. Other ways to customize the proof search are: strengthening the pure solver, and instructing Diaframe to merge some hypotheses. Merging hypotheses may be necessary to find relevant equalities or contradictions. The process of designing a user-provided
Figure 6. Data on verified examples. Rows correspond to files in the supplementary material [64]. Columns show number of lines of implementation of the program, annotation (specifications + invariants) and proof search customization. The format \( n/m \) stands for \( n \) lines in total, of which \( m \) lines consist of proof work. Proof search customization (i.e., user-provided hints) is always counted as proof work. In the hints column, notation \( h(c) \) stands for \( h \) distinct hints used for the proof, \( c \) of which were custom/user-provided. The time column displays the average verification time in minutes:seconds. The column \texttt{total} also includes all remaining Coq boilerplate, like \texttt{Import} statements.

Hint is generally as follows. First, run Diaframe until it gets stuck. Inspect the available hypotheses and goal, looking for a hypothesis that indicates a way to prove the left-most atom in the goal. Create and prove this new hint, and repeat.

Performance for failing verifications. One rarely gets the verification of these examples right in one go. It is therefore important to consider the performance of Diaframe when verification fails. In our artifact [64] one can find several examples that intentionally fail, obtained by changing the code, postcondition or omitting induction hypotheses. In all these cases, failing times were lower than the final verification time in Figure 6.

Differences between the examples across tools. We verify \texttt{bounded} for a parametric bound, whereas \texttt{Caper} and \texttt{Voila} fix the bound to 3. \texttt{Starling} verifies a static version of Peterson’s algorithm and a bounded reader-writers lock, whereas we verify a heap-allocated version.

Manual Iris proofs. When comparing with manual Iris proofs, we see that Diaframe takes care of most, if not all, of the proof work. Relatively easy examples like \texttt{spin} and \texttt{cas} are verified without manual proof work. For harder examples like \texttt{ticket} and \texttt{bag}, Diaframe saves more than 50 lines of proof work.

Starling. Starling [90] functions as a \texttt{proof outline checker}: the user has to supply the intermediate program states after each atomic step, and Starling will then verify whether this transition is valid. Starling is a standalone tool written in F#, and can use different backends as trusted oracles—the Z3 SMT solver [27], or GRASShopper for heap-based reasoning [73]. Its logic is based on the Views framework [29], which enables Starling to express various concurrent reasoning patterns into one core proof rule. This core proof rule produces a finite set of verification conditions for each atomic step, which can then be sent to the trusted oracle. This
efficient mapping of atomic steps to verification conditions, together with the ease of defining custom concurrent reasoning patterns, gives Starling’s proof automation its power. The downside of the relatively simple logic of Starling is reduced expressivity—it cannot prove functional correctness of e.g., the bag_stack. There is also no support for verifying method calls, preventing verification of clients.

Comparing our statistics to those of Starling, one can see that we usually require fewer lines of proof work. This is not surprising, as Starling is a proof outline checker, and thus requires a pre- and postcondition for every atomic operation. A notable exception to the smaller proof obligation is Peterson’s algorithm. Stating and proving the invariant for this algorithm in Iris turned out to be quite difficult, and it seems Starling’s constraint-based approach is a better fit here. In Figure 6, we counted postconditions of atomic operation that are not the last operation as proof work, as well as non-comment lines in program-specific external files.

**Caper.** Caper [31] is written in Haskell, and uses the Z3 SMT solver [27] as a trusted oracle. The target programs are written in a custom language, and the proof system is based on the CAP logic [30]. This logic contains shared regions (similar to Iris’s invariants) and guard algebras (similar to Iris’s ghost state/logical resources) to accommodate reasoning about fine-grained concurrency. The cornerstones of Caper’s proof automation are backtracking and abduction. These allow Caper to infer that regions should be opened when verifying the execution of a statement in a program. A failure to satisfy some precondition is used as an indication to reattempt the proof with opened regions.

When comparing Diaframe to Caper, we can see that Caper outperforms Diaframe in terms of proof work and annotation overhead. For one, their notations can give implementations and specifications of functions in one go. Caper’s proof automation is also simply more powerful—notably, it will ‘blindly’ open regions in the hope they help proving the goal. Although this makes Caper’s automation more powerful, it also makes it slow on failing examples as pointed out by Wolf et al. [91]. In these cases, Diaframe’s automation will simply stop at the point where it cannot make progress, while Caper will backtrack through all possible options. In the verification of clients, we outperform Caper because Diaframe’s verification is compositional—unlike Caper, we do not need to restate and re-verify a library to verify a client.

For Caper, the lines of proof work in Figure 6 consist of no-ops such as assert (cnt = 1 ? true : true), that are used to force case-splits in Caper’s proof engine.

**Voila.** Voila [91] is a proof outline checker for the TaDA logic [25]. Voila takes a user-provided proof outline, turns it into a proof candidate, then verifies this with Viper [65]. Like Caper, Voila uses regions and guard algebras for fine-grained concurrent reasoning. Some program statements need additional annotations containing the relevant reasoning steps, like opening regions. Voila’s automation is a combination of applying syntax-driven rules whenever possible, asking the user to provide key rules of the proof, and then using a set of heuristics to fill in gaps for nearly applicable rules.

In the examples in our benchmark, Diaframe usually requires fewer total lines, and fewer lines of user guidance than Voila. Again, this is not surprising, since like Starling, Voila is a proof outline checker. Voila also does not support all the guard algebras that Caper does. This prevents verification of e.g., the queue. However, Voila is capable of (and focused at) verifying TaDA-style logically-atomic specifications. While Iris supports these, Diaframe does not. For Voila, the lines of proof work in Figure 6 consist of explicit calls to open/close regions, and explicit uses of atomic specifications.

### 7 Related Work

There is a lot of work on non-automated verification [47, 50, 59] in foundational tools [3, 17, 39, 45, 67, 77]. We focus on related work in automated verification. Starling [90], Caper [31] and Voila [91] have been covered in §6.

**Steel.** Steel [36, 83] is a language for developing and verifying concurrent programs in a concurrent separation logic descendant of Iris [45], written in F* [82]. Similar to Diaframe, Steel designed a format to automate the application of certain rules. Their approach uses a notion of Hoare quintuples, and relies on a combination of SMT solving and AC-matching. Diaframe uses weakest preconditions, and avoids reasoning up to commutativity: the order in preconditions and invariants is relevant. Steel excels in automatically proving pure side conditions, leveraging F*’s native use of the Z3 SMT solver [27]. As listed in §8, our support for pure side conditions is rather weak, and would benefit from stronger pure automation. It is hard to compare Steel’s automation for fine-grained concurrency to ours, since Fromherz et al. [36] only covered a spinlock and a parallel increment.

**Verification in a weak-memory setting.** Summers and Müller [79] presented a prototype tool which can automatically verify fine-grained concurrent programs in a weak memory model. It works by encoding parts of separation logics for weak memory [33, 34, 88] into Viper [65], similar to Voila’s approach [91]. It would be interesting to extend Diaframe with support for weak memory using one of the Iris-based logics for weak memory [26, 49, 62].

**Bedrock.** Bedrock [21, 22] is a mostly-automatic foundational tool for verifying sequential programs in an assembly-like language. Its separation-logic based automation employs techniques that are somewhat similar to those of Diaframe. It tries to syntactically match hypotheses and goals, ‘crossing off’ hypotheses that appear directly in the goal. More involved reasoning steps, like updating ghost-state, require explicit annotations, and we expect that this would not give the amount of automation that Diaframe provides.
**RefinedC.** RefinedC [76] is a recent Iris-based tool for automatic and foundational verification of C programs. One of the main ingredients of RefinedC’s automation is the ‘separation logic programming language’ Lithium, which, like Diaframe, is based on ideas from linear logic programming. Lithium and Diaframe employ the same rules for introducing variables and hypotheses, prove separating conjunctions in a deterministic left-to-right fashion, and do not backtrack once a hint has been used. Lithium’s grammar is more restricted than Diaframe’s—it does not contain modalities, so it cannot handle complicated ghost state or Iris’s invariants. It is also targeted specifically at proving RefinedC’s typing judgments, while we target general Iris weakest preconditions. By encapsulating some concurrency reasoning in typing rules, RefinedC can support limited forms of fine-grained concurrency, like a spin-lock and a one-time barrier. RefinedC has stronger automation and simplification procedures for pure goals, focused at handling complicated sequential programs, which might be valuable for Diaframe in the future too.

**Other non-foundational verification tools.** Other automated verification tools are Verifast [10, 44], SmallfootRG [8, 16], and VerCors [69]. The automation of Verifast is very fast and requires little help for sequential code, but many annotations for fine-grained concurrent code compared to other tools. SmallfootRG is targeted at memory safety, thus cannot prove full functional correctness like Diaframe. Like Diaframe, Verifast and Smallfoot use automation by symbolic execution. An important difference is the use of a symbolic heap, which facilitates permission and value queries. We do not have this option in Iris, so instead of operating on the entire heap at once, we operate on a single hypothesis at a time. VerCors uses process-algebras in addition to separation logic to reason about fine-grained concurrent programs. This approach does lead to reduced expressivity, but has been shown to scale to interesting examples [70].

**Logic programming languages for linear and separation logic.** There is much prior work on linear logic programming [5, 19, 40, 43], from which our work has drawn inspiration. Like Diaframe, these works use a goal-directed proof-search procedure, and interpret connectives as proof-search instructions. They are usually restricted to the (linear) hereditary Harrop fragment of the logic, but enjoy completeness results on this fragment. Diaframe poses less restrictions on goals, but is necessarily incomplete. Inspired by focusing [1, 57] Diaframe first performs invertible operations.

**Separation logic solvers and bi-abduction.** The literature abounds with solvers for (first-order) separation logic [23, 55, 56, 72, 75, 84]. These usually focus on a specific set of atoms (e.g., the symbolic heap fragment [7]), or intricate recursive structures while enjoying completeness results. Diaframe is parametric in the set of atoms, but not able to handle recursive definitions without user-defined hints. Calcagno et al. [15] and Brotherston et al. [14] also use bi-abduction, but with a dual goal: shape-analysis, i.e., inferring specifications for programs. They present recursive rules and a decision procedure to solve the bi-abduction problem, but in a more confined separation logic.

### 8 Limitations and Future Work

We have introduced Diaframe—the first automated and foundational tool for verification of fine-grained concurrent programs. As the benchmarks in Figure 6 show, Diaframe is competitive with automatic non-foundational tools, but there are still plenty of directions for improvements.

A limitation of Diaframe is that it cannot handle goals that do not fit the grammar. In particular, there is no support for magic wands in invariants. Although these can be avoided in most cases, some examples remain out of reach—for example, the barrier verified by Jung et al. [45].

Some manual proof work is caused by the lack of support for recursive definitions, for which we would like to generate proof hints automatically. In this paper, we have focused on automating the separation logic part of the verification, but for larger examples we want to improve the automation and simplification procedures for pure conditions.

When our automation gets stuck on a goal, it can sometimes be unclear why this goal remains, and what happened before. This occurs most often in programs with multiple branches and/or invariants with disjunctions. We leave improving the user interaction in these cases for future work.

Since we use syntactic unification to drive automation, support for (general) indexing in an array is poor. Verification of data structures such as ring buffers seem like a challenge. It would be useful to develop appropriate hints for arrays.

The verification time of Diaframe is relatively slow. Although 18 out of 24 examples verify in under a minute, the barrier example is our slowest, taking 14 minutes. We think this can still be improved, and wish to investigate this.

Diaframe’s proof search strategy could, in principle, be used whenever goals can be rewritten into Diaframe’s entailment format. This can be done for logically-atomic specifications, and can also be done for ReLoc’s refinement judgment [37, 38]. However, both these types of goals present additional challenges for automatic verification—one of which is that there are multiple seemingly valid (and syntactically similar) ways to proceed with a proof. In future work, we wish to investigate whether this can be addressed.

### Acknowledgments

We thank Michael Sammler, Rodolphe Lepigre, and Jules Jacobs for useful discussions, and our shepherd Thomas Wies and the anonymous reviewers for their helpful feedback. This research was supported by the Dutch Research Council (NWO), project 016.Veni.192.259, and by generous awards from Google Android Security’s ASPIRE program.


