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
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—a 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.

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, 46, 65, 66, 72, 78, 79, 83–87], 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 [88] and Voila [89], generally use SMT [27] or separation-logic solvers [63, 71] 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 [43, 44, 46, 50], FCSL [75] 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, 49, 51, 58] 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 [78], 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 [43, 44, 46, 50]—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 optionally with a set of user-provided hints. The program and specification are turned into an Iris entailment that we prove.

Identifying good hints is one of the main challenges that we face. The proof rules of expressive logics like Iris (in particular, those 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 goals that appear in Iris proofs 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.

**Contributions.** We present Diaframe—a Coq library for Iris to automate the verification of fine-grained concurrent programs. Concretely, we make the following contributions:

- An entailment (§3) and hint format (§4) to capture goals and rules in Iris.
- A goal-directed proof search strategy for Iris that can be implemented with little backtracking in Coq (§5).
- A benchmark with proofs of correctness of 24 programs using fine-grained concurrency, and a comparison of proof-burden to Starling, Caper, and Voila (§6).

We start with two example verifications using Diaframe (§2). After covering our contributions (§3 to 6), we discuss related work (§7), and limitations and future work (§8).

## 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 \(\{P\} e \{\Phi\}\) in Iris, where \(P: iProp\) is a separation logic assertion and \(\Phi: Val \rightarrow iProp\) a separation logic predicate on values. The triple \(\{\} e \{\Phi\}\) 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 \(\Phi\) \(w\). 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 \(\text{SPEC } \{P\} e \{\gamma, \text{RET} v; Q\}\) for \(\{P\} e \{w. \exists \gamma. r = w \supset \gamma \supset Q\}\) to more succinctly specify return values. Note that \(r \gamma\) embeds a pure Coq proposition \(\phi\) into separation logic.

### 2.1 Verification of a Spinlock

Lines 1–8 in Figure 2 give the implementation of a spin lock in the ML-like language HeapLang—Iris’s default language [43]. The newlock method creates a new lock in 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 \(l\) 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 uses a regular store operation to put the lock back to the unlocked state (**false**). We do not need a CAS, since only one thread is in charge of releasing the lock.

Figure 1. Overview of the architecture of Diaframe. User input is marked in blue.
Let us now consider the specification of the lock methods, given in lines 15–26 in Figure 2. These specifications use the representation predicates \( \text{is\_lock} \ y \ \text{lk} \ R \) and \( \text{locked} \ y \) for locks [39, 78]. Here, \( \text{is\_lock} \ y \ \text{lk} \ R \) expresses that the lock at location \( \text{lk} \) protects assertions \( R \), and \( \text{locked} \ y \) expresses that the lock is in locked state. The ghost identifier \( y \) is used to tie these two representation predicates together.

Given an arbitrary assertion \( R \), the new lock method returns a value \( \text{lk} \), for which \( \text{is\_lock} \ y \ \text{lk} \ R \) holds. The assertion \( \text{is\_lock} \ y \ \text{lk} \ R \) 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 \( \text{locked} \ y \) that the lock is locked, and access to assertion \( R \). Contrary to \( \text{is\_lock} \ y \ \text{lk} \ R \), the assertion \( \text{locked} \ y \) is not duplicable, because at most one thread can hold the lock. To call release, we need to relinquish both \( \text{locked} \ y \) 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 \( \text{is\_lock} \ y \ \text{lk} \ R \) representation predicate is particularly flexible, since it is impredicative [78]—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 \( \ell \mapsto \alpha \) of separation logic expresses exclusive ownership of the location \( \ell \) with value \( \alpha \), we cannot just share it between multiple threads.

To reason about shared mutable state, we use Iris’s invariant assertion \( \Gamma \) \( N \), which says that there is a (shared) invariant with name \( N \) governing the resources satisfying Iris assertion \( \Gamma \). Invariants \( \Gamma \) \( N \) are duplicable, which means that the assertion \( \Gamma \) 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 \( \text{is\_lock} \ y \ \text{lk} \ R \). It says that a value \( \text{lk} \) is a lock if it is equal to some location \( 1 \), whose stored value is governed by an invariant \( \text{lock\_inv} \). Note that in Coq, we write \( \text{inv} \ N \ L \) for \( \Gamma \) \( N \). The invariant \( \text{lock\_inv} \) 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 \( \text{locked} \ y \) assertion are owned by the invariant.

The key ingredient for the verification of the spinlock is the ghost assertion \( \text{locked} \ y \), whose rules are:

\[
\begin{align*}
\text{locked-allocate} & \quad \text{locked-unique} \\
\vdash & \Rightarrow \exists y. \text{locked} \ y & \text{locked} \ y \Rightarrow \text{locked} \ y \vdash \text{False}
\end{align*}
\]

The first rule is used in the proof of new lock. It allows for the allocation of \( \text{locked} \ y \) with a fresh ghost name \( y \). This assertion is needed to establish the invariant by proving the right disjunct of \( \text{lock\_inv} \). The update modality \( \Rightarrow \) 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 \( \text{locked} \ y \) is a singleton—no two threads/resources can simultaneously satisfy this assertion. This means that the \( \text{locked} \ y \) assertion gives us information about the global state. In the proof of release, just before executing the store, the right disjunct of \( \text{lock\_inv} \) is contradictory because \( \text{locked} \ y \) 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 automatically. Other concurrent programs may use different ghost assertions, but all of these assertions have three types of rules: (a) allocation/creation rules, like \( \text{locked-allocate} \), (b) compatibility/interaction rules, like \( \text{locked-unique} \), and (c) mutation/update rules, of the form \( P \ast Q \vdash \Rightarrow R \ast S \). We will see some update rules in the next example.

### 2.2 Verification of an ARC

We will now verify a version of an Atomic Reference Counter (ARC), similar to the one verified by Starling [88] and the one used in the Rust standard library [52]. 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 \( \text{mk\_arc} \) 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 \( \text{clone} \) method increments the reference count with 1, using the atomic Fetch And Add (FAA) instruction, while \( \text{drop} \) decrements the reference count with 1. The unwrap method is like \( \text{drop} \) 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.

---

1 For readers familiar with Iris, we simply define \( \text{locked} \ y \triangleq \{ \ell \mapsto \text{lock} \ y \} \).
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 \( P : Q_p \rightarrow iProp \), where \( iProp \) is the type of Iris assertions, and \( Q_p \) is a set of permissions. Predicates \( P \) of this type must satisfy \( \forall q \in Q_p \mid q > 0 \). Predicates \( P \) must satisfy \( P q_1 \star P q_2 \vdash P(q_1 + q_2) \) to be called shareable (or fractional in Coq). An example of a shareable assertion is the fractional mapsto connective \( \ell \mapsto_{q} a \). 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 duplicable assertion \( \text{is}_{\text{arc}} \gamma v \) says that a value \( v \) is an ARC. The non-duplicable assertion \( \text{token-allocate} P \gamma \) indicates that a read-access reference to \( P \) has been recovered, i.e., that no read-access tokens \( P \gamma \) exist.

With these predicates at hand, the specification of \( \text{mk}_{\text{arc}} \) requires \( P1 \) (write-access) and returns a value that \( \text{is}_{\text{arc}} \gamma P \) 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, writes the spinlock, with the knowledge that \( \text{no}_{\text{tokens}} P \gamma \) exists. The unwrap method, when it terminates, guarantees retrieving write-access \( P1 \) and \( \text{no}_{\text{tokens}} P \gamma \).

Let us look at the definition of \( \text{is}_{\text{arc}} \gamma \) in lines 14–19 in Figure 3. Similar to \( \text{is}_{\text{arc}} \gamma \) in lines 14–19, we treat token and \( \text{no}_{\text{tokens}} P \gamma \) abstractly (see our appendix [62] 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.

Figure 3. Verification of an ARC in Diaframe.

Figure 4. Rules for the counter ghost assertions.
Similar to the spinlock, we define a value to be $\texttt{i}_\text{arc}$ if it is a location whose stored value is governed by an invariant. This invariant $\texttt{arc}_{\text{inv}}$ 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 $P \land z$. The counter $P \land z$ assertion states the knowledge that precisely $p \times z$ tokens currently exist—which matches what we want $\ell \mapsto p$ to mean.

To prove the specification of the count method, we use $\texttt{TOKEN-ALLOCATE}$, which allows us to establish the left disjunct of $\texttt{arc}_{\text{inv}}$. For proving the specification of count, we rely on $\texttt{TOKEN-INTERACT}$ to prove that the right disjunct of $\texttt{arc}_{\text{inv}}$ is contradictory. For the specification of $\texttt{clone}$, we again need $\texttt{TOKEN-INTERACT}$. When closing the invariant, we need to apply $\texttt{TOKEN-MUTATE-INCREMENT}$ at the right moment to change the obtained counter $P \land z$ to the required counter $P \land (p + 1)$. This also gives us the extra token that we need in the postcondition.

When proving $\texttt{drop}$, we need to use $\texttt{TOKEN-MUTATE-DELETE-LAST}$. Diaframe is not smart enough to figure out the required case distinction automatically. However, since Diaframe only performs limited backtracking, the automation simply stops when it encounters a goal it cannot make progress on. After manually performing the required case analysis, we resume the Diaframe automation with the $\texttt{StepsS}$ tactic, which then solves the goal.

The ghost assertions we use here (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 $\texttt{arc}_{\text{inv}}$, 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 $\texttt{drop}$.

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 ($\S 3.1$), as commonly done in interactive and automatic tools in proof assistants [20, 51, 74]. We then extend this approach with support for Iris’s invariant mechanism to verify fine-grained concurrent programs ($\S 3.2$). We conclude with an overview of the Diaframe entailment format and proof strategy ($\S 3.3$), which serves as a starting point for the description of our hint format ($\S 4$).

### 3.1 Goal-Directed Reasoning with WP

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

$$\{P\} \texttt{e} \{\Phi\} \equiv P \land \texttt{wp} \ e \{\Phi\}.$$ 

To get some intuition for the semantics of $\texttt{wp} \ e \{\Phi\}$, assume for a moment that $P$ and $Q$ are predicates on heaps (ignoring Iris’s ghost state and step-indexing), and $\Phi$ is a predicate on values and heaps. Entailment $P \land Q$ means that for every heap $h$, if $P \land h$ holds, then $Q \land h$ holds. The assertion $\texttt{wp} \ e \{\Phi\}$ describes the heaps for which execution of $e$ is safe (cannot get stuck), and if $e$ terminates with value $v$ and heap $h’$, then $\Phi \land h’$ 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 separation logic entailment. In particular, they give us access to Iris’s existing infrastructure [49, 51] for proving entailments. However, Iris’s primitive rules for weakest preconditions in Figure 5 are not syntax directed and can thus not be directly applied in an interactive or automatic proof search strategy. Throughout this section, we focus on transforming the rule $\texttt{WP-FAA}$ into a syntax-directed version. Recall that $\texttt{FAA}$ is used in the $\texttt{clone}$ and $\texttt{drop}$ methods of ARC ($\S 2.2$).

Suppose we are proving the following entailment:

$$\Delta \vdash \texttt{wp} \ {\texttt{FAA} \ \ell \ {\texttt{v}_2 \ {\texttt{v}_3}}} \ {\texttt{v}_1}.$$ 

(From now on, we will often put an environment $\Delta$ before the turnstile. The environment $\Delta$ is a list of assertions $P_1, \ldots, P_n$, for which $\Delta \vdash Q$ if $P_1 \land \cdots \land P_n \vdash Q$.)

We want to prove this entailment by applying $\texttt{WP-FAA}$, but we are not yet in shape to do so. That is because $\Delta$ will typically not be just $\ell \mapsto z_1$, and $\Phi$ will typically not be the precise postcondition of $\texttt{WP-FAA}$. Hence, to apply ‘small footprint’ specifications like $\texttt{WP-FAA}$ we need to find a ‘frame’ $R$ and a value $z_1$, such that $\Delta \vdash R \land \ell \mapsto z_1$. We can then use a combination of $\texttt{WP-FAA}$, $\texttt{WP-FRAME}$ and $\texttt{WP-MONO}$, to transform our entailment into $R \land \ell \mapsto (z_1 + z_2) \land \Phi z_1$.

Instead of having to determine the frame $R$ in advance, one can construct an alternative rule for goal-directed reasoning,
which will be easier to apply automatically:

\[
\text{wp-faa-ramify}
\Delta \vdash \ell \mapsto z_1 * (\forall \nu. (\nu = ?z_2 \land \ell \mapsto (?z_2 + z_3)) \rightarrow \Phi \nu) \\
\Delta \vdash \text{wp} (FAA \ell z_2) \{\Phi\}
\]

In this shape, the rule is an instance of the ramified frame rule [20, 40]. Note that we have put a question mark in front of \(z_1\) to signify that \(z_1\) will be an existential variable (evar) at rule application—we should be able to find a \(z_1\) for which this is provable, but do not yet know which one it will be. When we find an hypothesis of shape \(\ell \mapsto z\) in \(\Delta\), we can unify \(z_1\) with \(z\) and continue.

We can generalize the rule \text{wp-faa-ramify} to any Hoare-style specification of an expression \(e\):

\[
\text{sym-ex}
\{P\} e \{\Psi\}
\Delta \vdash \text{wp} \{K[v] \{\Phi\}\}
\Delta \vdash \text{wp} K[e] \{\Phi\}
\]

This rule additionally incorporates Iris’s rule \text{wp-bind}, which allows the expression \(e\) to appear inside a call-by-value evaluation context \(K\), instead of at the top-level.

Supposing we can prove separating conjunctions, \text{sym-ex} gives rise to a symbolic-execution based proof search strategy for straight-line sequential code. Suppose our goal is \(\Delta \vdash \text{wp} \{\Phi\}\). If \(e\) is a value \(v\), apply \text{wp-value} and prove \(\Delta \vdash \Phi v\). Else, find an evaluation context \(K\) and subexpression \(e’\) with \(e = K[e’]\), and a specification \(\{P\} e’ \{\Psi\}\). Apply \text{sym-ex}, prove the separating conjunction, introduce variables, 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 \text{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:

\[
\text{inv-open-wp}
\Delta, [L]^N, \triangleright L \vdash \text{wp}_{E|N} e \{v, \triangleright L * \Phi v\} \quad \text{atomic } e \quad N \subseteq E
\]

\[
\Delta, [L]^N \vdash \text{wp}_{E} e \{\Phi\}
\]

This rule is quite a mouthful, so let us go over it step by step. First, to deal with invariants, weakest preconditions in Iris \text{wp}_{E} e \{\Phi\} 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 \(\top\), meaning all invariants can still be opened.

Suppose that we have an invariant [L]^N, and are verifying an atomic expression \(e\). Rule \text{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 postcondition 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, 64] is needed for technical reasons caused by the fact that invariants are impredicative [44, 78], i.e., the resource \(L\) in an invariant can be any resource, including invariants and weakest preconditions. Dealing with later modalities requires some additional bookkeeping, which Diaframe deals with automatically, but we gloss over in this paper.

We now show why our \text{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 clone method of the ARC (§2.2). The challenge of verifying this method is that the \(\ell \mapsto _\text{clone}\) we need as part of the precondition for FAA is not in the proof context, but in an invariant \([z: \mathbb{Z}]\). When we apply \text{sym-ex} eagerly, we lose the ability to open invariants using \text{inv-open-wp}.

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

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

\[
\text{sym-ex-fupd-exist}
\forall \mathbb{X}. \{P\} e \{\Psi\} \quad \text{atomic } e \lor E_1 \Rightarrow E_2
\]

\[
\Delta \vdash E_1 \rightarrow E_2 \exists \mathbb{X}. P * \big(\forall \nu. \Psi \nu \lor \exists \mathbb{Z}. wp_{E_1} K[w] \{\Phi\}\big)
\]

\[
\Delta \vdash \text{wp}_{E_1} K[e] \{\Phi\}
\]

This rule contains Iris’s fancy update modality \(\triangleright E_1\), and a quantified Hoare triple \(\forall \mathbb{X}. \{P\} e \{\Psi\}\).

The fancy update modality \(\triangleright E_1\) is used in Iris’s definition of weakest preconditions, and is the component that makes opening invariants possible. Semantically, \(\triangleright E_1\) means: assuming all invariants with names in \(E_1\) hold, then \(P\) holds and additionally all invariants with names in \(E_2\) hold. To work with the fancy update, Iris has the following rules:

\[
\text{inv-open-fupd}
\Delta \vdash L^N \triangleright E_1 N \triangleright L * (\triangleright L * \text{wp}_{E_1} N \{\Phi\})
\]

\[
\Delta, [L]^N \vdash \text{wp}_{E} e \{\Phi\}
\]

The \text{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 (\(\triangleright L \leftarrow E_1 N \rightarrow \top\)). The rule \text{fupd-elim} allows us to compose fancy updates, and by combining \text{fupd-fupd} and \text{fupd-intro} we can introduce the last fancy
When encountering an existential, the Diaframe proof search strategy delays the existential but (wrongly) performs an eager instantiation of the existential invariants through a combination of the rules BUPD-ELIM and INV-OPEN-FUPD. Our new rule is strictly stronger than the rule SYM-EX from §3.1—the update modalities can simply be instantiated using BUPD-FUPD and 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 SYM-EX-FUPD-EXIST wrongly by instantiating existentials eagerly in a goal that arises during the verification of an FAA in ARC (§2.2):

$$\ell \rightarrow z, z, \ldots \vdash_1 \frac{1}{N} \overset{\exists E}{\ell} \rightarrow ?z_1 \ldots$$

$$\vdash (\exists (z : Z), \ell \rightarrow z \ast j z), \ldots \vdash_1 \frac{1}{N} \overset{\exists E}{\ell} \rightarrow ?z_1 \ldots$$

One should read this proof derivation from bottom to top. When encountering an FAA, we apply SYM-EX-FUPD-EXIST, but (wrongly) perform an eager instantiation of the existential $z'$ with an evar $?z_1$. Then we use INV-OPEN-FUPD and FUPD-ELIM to open the invariant. The final step uses some properties of the later modality to eliminate the existential and the later around $\ell \rightarrow z$. One might think we are now done: just unify $?z_1$ with $z$ and $?c'$ with $\tau \setminus N$, and continue! However, this is not sound—the evar $?z_1$ cannot be unified with $z$, since $z$ was introduced after $z_1$. Stated in other words, we could not have chosen $z_1$ to be equal to $z$, since at that point $z$ was not in our context. To correctly deal with existentials, the Diaframe proof search strategy delays the instantiation of existentials.

### 3.3 Overview of the Diaframe Strategy

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

1. If the goal is $\Delta \vdash \forall x. \mathcal{G}$ or $\Delta \vdash U \Rightarrow \mathcal{G}$, introduce the $\forall$ or $\Rightarrow$. Then “clean” the hypothesis $U$ by (a) eliminating separating conjunctions, disjunctions, and existentials, (b) moving pure assertions $\Gamma \phi \gamma$ into the Coq context, (c) merging assertion, e.g., $\ell \rightarrow \gamma \omega$ and $\ell \rightarrow \gamma \omega$ become $\ell \rightarrow \rho_{\omega} \gamma$ and $\omega = \omega$, (d) deriving contradictions, e.g., using \texttt{LOCKED-UNIQUE}.
2. If the goal is $\Delta \vdash \omega \wedge \{P\}$, with $\omega$ a value, continue with $\Delta \vdash \gamma \phi \zeta \Phi$.
3. If the goal is $\Delta \vdash \omega \wedge K[e] \{\Phi\}$, use our new symbolic execution rule SYM-EX-FUPD-EXIST. Our new goal has the shape $\Delta \vdash \overline{e_1} \overline{e_2} \overline{x} \exists \gamma. L \wedge G$.
4. If the goal is $\Delta \vdash \overline{e_1} \overline{e_2} \overline{x} \exists \gamma. L \ast G$, use associativity of separating conjunction to rewrite it into $\overline{e_1} \overline{e_2} \overline{x} \exists \gamma. A + G'$ where $A$ is an atom. Pure conditions $\overline{r} \phi \gamma$ that appear in the process are solved with Coq tactics like \texttt{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 one of the above entailment formats so the strategy can make repeated progress. When operating on entailments of format $\Delta \vdash \overline{e_1} \overline{e_2} \overline{x} \exists \gamma. L \ast G$, 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—no 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, 41] and recent work on the separation logic programming language Lithm [74]. The fundamental difference is that we do not operate on top-level connectives, but on those 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 fails short.

There are two ways how hints can be selected. \textit{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. \textit{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 \textit{hint format} (§4.1) that is inspired by the technique of bi-abduction [15]. Aside from the base hints (§4.2), Diaframe provides \textit{recursive hints} to close the base hints under connectives like invariants, magic wands, and separating conjunctions (§4.3).

#### 4.1 Bi-Abduction Hints

The \textit{hint format} of Diaframe is as follows:

$$H \vdash \overline{y}; \overline{x} \vdash \overline{e_1} \overline{e_2} \overline{x} \exists \gamma. L \sim; \overline{u} \overline{v} \rightarrow \overline{e_2} \overline{e_1} \overline{x} \exists \gamma. A \ast U \vdash \forall \overline{y}, \overline{u}, \overline{v}.$$
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 \( \epsilon_1 \) for a last-resort hint. The assertion \( \epsilon_1 \) 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] between 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 \( H \Rightarrow \text{antiframe} \vdash A \Rightarrow \text{frame} \). 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:

\[
\text{BIABD-HINT-APPLY}
\]

\[
H \triangleright [\bar{y}; L] \vdash \left[ \epsilon_1 \triangleright \mathcal{E}_2 \right] : \bar{x}; A \vdash U \quad \Delta \vdash \mathcal{E}_1 \triangleright \exists \bar{y}. L \quad (\forall \bar{x}. U \Rightarrow G)
\]

\[
\Delta, H \vdash \epsilon_1 \triangleright \mathcal{E}_2 \triangleright \exists \bar{y}. A \vdash G
\]

The Diaframe implementation will go over the hypotheses \( H \) in the context \( \Delta \) from left to right (with \( \epsilon_1 \) last) until it finds a hint \( H \triangleright [\bar{y}; L] \vdash \left[ \epsilon_1 \triangleright \mathcal{E}_2 \right] : \bar{x}; A \vdash U \) 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 \textsc{token-mutate-decr} (which is used to verify the drop method of ARC in §2.2) into the following hint:

\[
\text{counter } P \gamma p \triangleright [\vdots; \text{token } P \gamma y \Rightarrow \gamma p > 1] \vdash \left[ \epsilon_1 \triangleright \mathcal{E} \right] : \vdots; \text{counter } P \gamma (p - 1) \triangleright [\text{True}]
\]

If we use \textsc{biabd-hint-apply} with this hint, we get:

\[
\Delta \vdash \mathcal{E}_1 \triangleright \mathcal{E} \text{ token } P \gamma y \Rightarrow \gamma p > 1 \triangleright \text{ (True } \Rightarrow G)
\]

\[
\Delta, \text{counter } P \gamma p \triangleright \mathcal{E}_1 \triangleright \text{counter } P \gamma (p - 1) \triangleright G
\]

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

**Example 2: Invariant allocation.** In Iris, invariants are allocated using the rule \( \triangleright L \vdash \mathcal{E}_1 \triangleright [L]^N \), which we transform into the following hint:

\[
\epsilon_1 \triangleright [\vdots; \triangleright L] \vdash \left[ \epsilon_1 \triangleright \mathcal{E} \right] : \vdots; \left[ L^N \triangleright \left[ L^N \right] \right]
\]

Due to the \( \epsilon_1 \), this is a last-resort goal-directed hint. We do not make it hypothesis directed, because \( \triangleright L \) will usually not be precisely in the context. Since invariants are duplicable we give back \( [L]^N \) in the residue, so that it can be used again.

**Example 3: Ghost state allocation.** We transform the rule \textsc{locked_allocate} (which is used to verify the newlock method in §2.1) into the following hint:

\[
\epsilon_1 \triangleright [\vdots; \text{True}] \vdash \left[ \epsilon_1 \triangleright \mathcal{E} \right] : \gamma; \text{locked } y \triangleright [\text{True}]
\]

Due to the \( \epsilon_1 \), this is again a last-resort goal-directed hint. That is simply because the rule has no premise.

**Example 4: Point-to assertion.** We have specific hints for HeapLang’s fractional points-to assertion \( \ell \rightarrow_q v \):

\[
\ell \rightarrow_q v_1 \triangleright [\vdots; \gamma v_1 = v_2 \triangleright] \vdash \left[ \epsilon_1 \triangleright \mathcal{E} \right] : \vdots; \ell \rightarrow_q v_2 \triangleright [\text{True}]
\]

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:

\[
\ell \rightarrow_q v_1 \triangleright [\vdots; \gamma v_1 = v_2 \triangleright] \vdash \left[ \epsilon_1 \triangleright \mathcal{E} \right] : \vdots; \ell \rightarrow_q v_2 \triangleright [\gamma v_1 = v_3 \triangleright]
\]

This hint applies if the fraction \( q_2 \) in the goal is bigger than the fraction \( q_1 \) in the hypothesis, and hence has the side condition \( \ell \rightarrow_q v_3 \). Note that \( q_3 \) is existentially quantified, meaning that the side condition can be established for any value. This is sound by the agreement property of \( \Rightarrow \). 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:

\[
\ell \rightarrow_q v_1 \triangleright [\vdots; \gamma v_1 = v_2 \triangleright] \vdash \left[ \epsilon_1 \triangleright \mathcal{E} \right] : \vdots; \ell \rightarrow_q v_2 \triangleright [\gamma v_1 = v_3 \triangleright]
\]

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 \( L_1 \rightarrow U \) 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:

\[
\ell \rightarrow_q v_1 \triangleright [\vdots; \gamma v_1 = v_2 \triangleright] \vdash \left[ \epsilon_1 \triangleright \mathcal{E} \right] : \vdots; \ell \rightarrow_q \ell_1 \triangleright [\gamma v_1 = v_3 \triangleright]
\]

\[2\text{In the implementation, this rule is a consequence of other recursive rules.}\]
This rule states that there is a hint from an invariant \( \prod_{l_1}^N \) to a goal \( A \), if there is a hint from the contained assertion \( l_1 \) to that atom. We get \( \mathcal{F} \subseteq \mathcal{E} \) as an additional side condition, and receive the closing update \( (\cdot l_1 \rightarrow \mathcal{E} \setminus l_1 \rightarrow \mathcal{F}) \) as the residue. Similar to \( e_1 \), 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 [62]):

\[
\begin{align*}
\text{atoms} & \quad A ::= \mathop{wp} e \{ v. L \} \mid \chi \mid \prod_{l_1}^N \cdots \\
\text{left-goals} & \quad L ::= \mathop{\mathcal{E}}^{\phi} A \mid A \land L \mid \exists x. L \\
\text{unstructured} & \quad U ::= \mathop{\mathcal{E}}^{\phi} A \mid A \lor U \mid \exists x. L \\
\text{extended} & \quad H ::= e_1 \mid U \\
\text{clean hypotheses} & \quad H_C ::= A \mid \forall x. U \land L \lor U \land \mathop{\mathcal{E}}^{e_2} U \\
\text{environments (1)} & \quad \Gamma ::= \emptyset \mid \Gamma, x \mid \Gamma, \phi \\
\text{environments (2)} & \quad \Lambda ::= \emptyset \mid \Lambda, H_C \mid \Lambda ::= \Lambda, e_1 \\
\text{goals} & \quad G ::= \forall x. G \mid U \Rightarrow G \mid \mathop{wp} e \{ v. L \} \\
& \quad \mid \mathop{\mathcal{E}}^{e_1} \mathcal{E}^{e_2} L \mid \mathcal{E}^{e_1} \mathcal{E}^{e_2} \exists \mathcal{X}. L \lor G
\end{align*}
\]

The entailments we wish to solve are of the form \( \Gamma; \Delta \vdash G \). The atoms \( A \) by default only consist of weakest preconditions \( \mathop{wp} e \{ v. L \} \), the marker \( \chi \) (§4.3) and invariants \( \prod_{l_1}^N \). 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 locked \( \gamma \). The definition of \( \Delta \) explicitly sets the last-resort marker \( e_1 \) 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
\[
\mathcal{E}^{e_2} \exists \mathcal{X}. L \lor R \quad \text{in} \quad G \quad \text{is a ‘synthetic’ representation of}
\]
\[
\mathcal{E}^{e_2} \exists \mathcal{X}. L \lor R\quad \text{with the condition} \quad \mathop{\mathcal{F}}(L) = \mathcal{X}.
\]

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 \( \mathcal{E}^{e_2} \exists v_1 \cdot v_2, l_1 \Rightarrow v_1 \lor v_2 \Rightarrow v_1 \) into
\[
\mathcal{E}^{e_2} \exists v_1 \cdot v_1 \Rightarrow v_1 \lor v_2 \Rightarrow v_1.
\]

#### 5.2 The Proof Search Strategy

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

1. \( G = \forall x. G' \); Continue with \( \Gamma, x \vdash G' \).
2. \( G = U \rightarrow G' \); Case analysis on \( U \):
   a. \( U = \mathcal{E}^{\phi} \); Continue with \( \Gamma, \phi \vdash G' \).
   b. \( U = (U_1 \lor U_2) \); Continue with \( \Gamma; \Delta \vdash U_1 \lor U_2 \rightarrow G' \).
   c. \( U = (\exists x. L) \); Continue with \( \Gamma, \forall x. (L \rightarrow G') \).
   d. \( U = H_C \); Continue with \( \Gamma; \Delta, H_C \vdash G' \).
3. \( G = \mathop{wp} e \{ v. L \} \):
   a. If \( e \) is a value \( w \), continue with \( \Gamma; \Delta \vdash \top \rightarrow \top \mathcal{L}[w/v] \).
   b. Else, find a \( K \) and \( e' \) with \( e = K[e'] \), and quantified specification \( \forall w. (L) \vdash (\forall w. L' \rightarrow \mathcal{E}^{e'}) \).
   c. \( \Gamma; \Delta \vdash \mathcal{F}(L \land (\mathcal{E}^{e'}[w/v]) \).

4. \( G = e_1 \rightarrow e_2 \); We consider the following cases:
   a. If the modality \( e_1 \rightarrow e_2 \) is not introducible, continue with \( \Gamma; \Delta \vdash e_1 \rightarrow e_2 \).
   b. \( \mathcal{E}^{e_1} \rightarrow e_2 \); Continue with \( \Gamma, \Delta, H_c \vdash e_2 \).
   c. \( \mathcal{E}^{e_1} \rightarrow e_2 \); Continue with \( \Gamma, \Delta, H_c \vdash e_2 \).
   d. \( \Gamma, \Delta \vdash e_1 \rightarrow e_2 \); True in all other cases.
   e. \( G = \mathcal{E}^{\phi} \rightarrow e_2 \); Case analysis on \( U \):
      a. \( \Gamma = \mathcal{E}^{\phi} \); Check that \( e_1 \rightarrow e_2 \) is introducible, and try to solve \( \phi[L/x] \).
      b. \( L = \mathcal{E}^{\phi} \lor \mathcal{E}^{\phi} \); Check that \( e_1 \rightarrow e_2 \) is introducible, and try to solve \( \phi[L/x] \).
      c. \( \Lambda = \emptyset \); Continue with \( \Gamma; \Delta \vdash \mathcal{E}^{e_1} \rightarrow e_2 \).
      d. \( \Lambda = \exists y. L \); Continue with \( \Gamma; \Delta \vdash \mathcal{E}^{e_1} \rightarrow e_2 \).
      e. \( \mathcal{E}^{\phi} \rightarrow e_2 \); Then continue with \( \Gamma, \Delta \vdash \mathcal{E}^{\phi} \rightarrow e_2 \).

In the above, we say that \( e_1 \rightarrow e_2 \) is introducible, if \( e_2 \) can be unified with \( e_1 \). Note that item 3b is SYM-EX-FUPD-EXIST (§3) and item 5d is BIABD-HINT-APPLY (§4). We have omitted steps in the introduction of magic wands to merge hypotheses and to detect incompatibilities. For example, if we introduce locked \( \gamma \) and already have a locked \( \gamma \) in our context, we obtain False by LOCKED-UNIQUE. We have also omitted the bookkeeping required to deal with Iris’s later modality (\( \top \)).

#### 5.3 Extending Diaframe with Disjunctions

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

To offer automation for some goals with disjunctions, we provide an extension of Diaframe. When introducing a disjunction \( \Delta \lor U_1 \land U_2 \rightarrow G \) into the context, continue with goals \( \Delta \lor U_1 \rightarrow G \) and \( \Delta \lor U_2 \rightarrow G \) by disjunction elimination. When proving \( \Gamma; \Delta \vdash \mathcal{E}^{\phi} \rightarrow e_2 \); (\( \mathcal{E}^{\phi} \lor U_1 \land U_2 \) \rightarrow G)
(and symmetrically), check if we can prove \( \lnot \phi \), and if so, continue with the simpler goal \( \Gamma \vdash \Pi \phi x_1 \ldots x_n \exists \mathbf{x} \mathcal{L} \vdash G \). 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 allow 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. 14,000 lines of Coq code, built on top of Iris. We use Coq’s type class mechanism [76] 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 iSteps$.  

This tactic can be used to prove specifications entirely, and as part of interactive proofs in the Iris Proof Mode [49, 51]. Diaframe comes equipped with 4 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 Caper [31], Starling [88] and Voila [89], 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. The examples that we have verified and their statistics can be found in Table 1. 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 of Diaframe 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 Caper, 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) Caper 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 Table 1. Diaframe can verify 7 of the examples without any help from the user. Averaged over all examples, we require about half a line of manual proof per line of implementation (465 lines of proof for 915 lines of implementation). The highest proof work is in the verification of the Michael-Scott queue [61], requiring 51 lines of proof for an implementation of 37 lines.

We shortly discuss differences between the examples across the tools. Our bounded_counter is parametric in the bound, whereas Caper and Voila verify it for a fixed bound of 3. Starling verifies a static version of Peterson’s algorithm and the bounded reader-writers lock, whereas we verify a heap-allocated version. The verification of the stack and queues in Diaframe requires (much) more proof search customization than the other examples. This is because they require a recursive definition specifying when a location is a concurrent stack/queue. Diaframe has no native support for recursive definitions yet, so some custom hints need to be provided.

### 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 spin_lock and cas_counter are verified without manual proof work. For harder examples like ticket_lock and bag_stack, Diaframe saves more than 60 lines of proof work.

### Starling

Starling [88] functions as a 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 [71]. 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 Starlings 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 Table 1, 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

---

Draft, November 23, 2021, Ike Mulder, Robbert Krebbers, and Herman Geuvers
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 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. [89]. 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 Table 1 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 [89] 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 [63]. 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 Table 1 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 [45, 48,
57] in foundational tools [3, 17, 37, 43, 65, 75]. We focus on
related work in automated verification. Starling [88], Caper
[31] and Voila [89] have been covered in §6.

Steel. Steel [36, 81] is a language for developing and veri-
fying concurrent programs in a concurrent separation logic
descendant of Iris [43], written in F* [80]. 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 invari-
ants 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 condi-
tions is rather weak. 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 [77] presented a prototype tool which can automatic-
ically verify fine-grained concurrent programs in a weak
memory model. It works by encoding parts of separation
logics for weak memory [33, 34, 86] into Viper [63], similar
to Voila’s approach [89]. It would be interesting to extend
Diaframe with support for weak memory using one of the
Iris-based logics for weak memory [26, 47, 60].

Bedrock. Bedrock [21, 22] is a foundational tool aimed at
verifying sequential programs in an assembly-like language.
It is mostly automatic and also based on separation logic. Its
automation employs techniques that are somewhat similar
to those of Diaframe, in that it tries to syntactically match
hypotheses and goals, ’crossing off’ hypotheses that appear
directly in the goal. Since it is focused at verifying sequential
programs, Bedrock’s language does not include primitives
for fine-grained concurrency.

RefinedC. RefinedC [74] is a recent Iris-based tool for au-
amatic and foundational verification of C-code. One of the
main ingredients of RefinedC’s automation is a ’separation
logic programming language’ called Lithium, which, like
Diaframe, is based on ideas from linear logic programming.
Lithium and Diaframe both 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 re-
stricted 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 judg-
ments, while we target general Iris weakest preconditions. By
encapsulating some concurrency reasoning in typing rules,
RefinedC can support limited forms of fine-grained concur-
rency, like a spin-lock and a one-time barrier. RefinedC has
stronger automation and simplification procedures for pure
goals, focused at handling complicated sequentical programs,
but which might be valuable for Diaframe in the future too.

Other non-foundational verification tools. Other au-
tomated verification tools are Verifast [10, 42], SmallfootRG
[8, 16], and VerCors [67]. The automation of Verifast is very
fast and requires little help for sequential code, but requires
lots of annotations for fine-grained concurrent code com-
pared to other tools. SmallfootRG is targeted at verifying
memory safety, thus cannot prove full functional correctness
like Diaframe. VerCors uses process-algebras in addition to
separation logic to reason about fine-grained concurrent pro-
grams. This approach does lead to reduced expressivity, but
has been shown to scale to interesting examples [68].

Logic programming languages for linear and separa-
tion logic. There is much prior work on linear logic pro-
gramming [5, 19, 38, 41], 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 complete-
ness results on this fragment. Diaframe poses less restrictions
on goals, but is necessarily incomplete. Inspired by focusing
[1, 55] Diaframe first performs invertible operations.

Separation logic solvers and biabduction. The litera-
ture abounds with solvers for (first-order) separation logic
[23, 53, 54, 70, 73, 82]. These usually focus on a specific set
of atoms (often a variant of the symbolic heap fragment [7]),
or target intricate recursive structures, while still enjoying
completeness results. Our approach is parametric in the set
of atoms, but not able to handle recursive definitions without
custom hints. An idea for future work is to investigate what
we could learn from these tools.

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 Table 1 show, Diaframe is competitive with automatic non-foundational tools, but there are still plenty of directions for improvements.

Some manual proof work is caused by the lack of support for recursive definitions, for which we want 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.

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 15 out of 24 examples verify in under a minute, the barrier example is our slowest with a whopping 25 minutes. We think this can be improved, and wish to properly profile and investigate this slowdown.

Acknowledgments

This research was supported in part by the Dutch Research Council (NWO), project 016.Veni.192.259, and in part by generous awards from Android Security’s ASPIRE program and from Google Research.

References


