Raven: An SMT-Based Concurrency Verifier

Ekanshdeep Gupta[1], Nisarg Patel*[2], and Thomas Wies[1]

[1] New York University
[2] Certora

Abstract This paper presents Raven, a new intermediate verification language and deductive verification tool that provides inbuilt support for concurrency reasoning. Raven's meta-theory is based on the higher-order concurrent separation logic Iris, incorporating core features such as user-definable ghost state and thread-modular reasoning via shared-state invariants. To achieve better accessibility and enable proof automation via SMT solvers, Raven restricts Iris to its first-order fragment. The entailed loss of expressivity is mitigated by a higher-order module system that enables proof modularization and reuse. We provide an overview of the Raven language and describe key aspects of the supported proof automation. We evaluate Raven on a benchmark suite of verification tasks comprising linearizability and memory safety proofs for common concurrent data structures and clients as well as one larger case study. Our evaluation shows that Raven improves over existing proof automation tools for Iris in terms of verification times and usability. Moreover, the tool significantly reduces the proof overhead compared to proofs constructed using the Iris/Rocq proof mode.

1 Introduction

We present Raven, a new language and SMT-based tool for deductive verification of concurrent programs. Raven is designed as an intermediate verification language (IVL) that can be used for building *front-end* verifiers for other programming languages.

Several existing SMT-based IVLs such as Why3 [13], Boogie [3], and Viper [37] have been used successfully to build concurrent program verifiers. However, none of these IVLs provide direct support for reasoning about concurrency. As a consequence, front-end tool developers are left with building their own concurrency support from scratch [6, 43, 49]. This requires an extra layer of encoding that can be a source of subtle soundness bugs [44].

Raven aims to fill this gap by providing inbuilt support for concurrency that is based on a solid theoretical foundation, yet builds on the lessons learned from existing IVL developments. Raven's design allows us to do the soundness reasoning for concurrency once at the IVL level rather than having to do it anew for each verification frontend. This argument extends to implementation bugs: eliminating redundancy in the verification tool pipeline reduces sources of potential soundness bugs.

Raven's meta-theory is based on the concurrent separation logic Iris [20, 22]. Iris is parametric in the underlying programming language. It is also very expressive due to its higher-order features that can be used to define new verification methodologies and

* Work started while at NYU

prove them sound within the logic itself. Thus, the logic can be thought of as an IVL in its own right. In fact, several efforts exist to build verification tools for real-world programming languages on top of Iris, including C [32] and Rust [15].

So why is there a need for a new tool? The answer is twofold. First, Iris is complex. This complexity is a key prerequisite to some of the logic's most impressive success stories [19]. But on the flip side, this complexity comes at the cost of a steep learning curve. This is despite the fact that many of Iris's advanced features are usually not needed for more mundane program verification tasks. In fact, in such cases, they can get in the way of a seamless user experience. Raven's design therefore restricts Iris's features to a subset that is geared towards these common use cases. The goal of this design is to make the logic more accessible and aid proof automation.

This brings us to the second point. The existing tooling effort in the Iris ecosystem has centered on its Rocq mechanization and accompanying proof mode [26]. While the proofs constructed with these tools provide strong foundational correctness guarantees, they also tend to be much more laborious compared to those done using SMT-based deductive verifiers. Recent efforts, notably the Diaframe project [34, 35] aim to provide better automation for Iris within Rocq. Diaframe already achieves impressive results and is able to almost fully automate linearizability proofs of small but intricate data structures. However, using this tool effectively requires a deep understanding of not just Iris, but also the intricacies of its low-level mechanization in Rocq. Moreover, Diaframe does not help with the automation of theory reasoning.

Overview and Contributions. In §2 we introduce Raven's core features. These include (i) a basic imperative programming language with concurrency primitives, (ii) a higher-order module system that aids proof modularization and reuse, (iii) an Iris-based specification language with invariants for thread-modular reasoning about shared state, and (iv) user-definable ghost state based on Iris's notion of resource algebras. In §3 we provide an overview of the proof automation support for these core features and discuss aspects of the SMT encoding. §4 covers some of Raven's advanced features such as atomic contracts for modular reasoning about linearizability, support for iterated separating conjunctions, and heuristics for dealing with quantifier alternations in specifications. In §5 we discuss implementation details and our evaluation of Raven. Finally, §6 discusses related work in more detail and §7 concludes.

We end here with a brief summary of the key findings of our evaluation. First, we compare Raven against Diaframe on Diaframe's verification benchmark suite of concurrent data structures and accompanying client programs. The takeaway is that Diaframe generally requires fewer proof annotations for these benchmarks. However, Raven's verification times are 1-2 orders of magnitude faster, resulting in a more agile interactive user experience. To assess the scalability and usability of the tools, we also reimplemented a more complex existing Iris linearizability proof in Raven. The original proof consists of several thousand lines of Rocq code and was done without the help of Diaframe. Compared to the original proof, Raven achieves a reduction in proof overhead by a factor of 12, and runs 8 times faster. We also used Diaframe to automate aspects of the original Iris development but quickly ran into usability issues with the tool's current proof automation support. Finally, we also compare Raven with Viper, another SMT-

based IVL, on a subset of sequential examples and find the two tools' performance to be comparable.

In summary, we believe that Raven provides a solid foundation for the development of concurrent program verifiers and is a useful verification tool in its own right.

2 Overview of Core Functionality

We introduce Raven's language and core functionality by using the verification of a ticket lock implementation as an illustrative example.

Module System. Raven programs are organized as modules. The members of a module include data type definitions, pure functions and values defined over these data types, fields that introduce (shared) heap-allocated state, procedures that perform computations over this state, as well as additional members related to specification and verification. The module system also supports nested and higher-order modules (or *functors*) that take other modules as parameters. The types of functor parameters are specified by *interfaces*. An interface is like a module but is allowed to have *abstract* members that are declared but not yet defined.

Figure 1a shows the signature of the interface `Lock` that specifies the operations supported by a lock instance `l`. The inbuilt type `Ref` of `l` represents references to heap locations. `Lock` abstracts over a module `R` that implements the interface `LockResource`. This interface declares the abstract predicate `resource(r)`, which represents a shared resource that can be protected by a lock. The value `r` of the representation (or **`rep`**) type `T` is used to uniquely identify this resource. Each module and interface can have at most one **`rep`** type member. A module is identified with its **`rep`** type when its name is used in a context that expects a type expression. For instance, the occurrence of `R` on §2 implicitly expands to `R.T`.

Specification Language. Raven provides a rich specification language based on concurrent separation logic with (abstract) predicates, shared invariants, user-definable ghost state, axioms, lemmas, etc. In our example, the axiom `LockResource.exclusive(r)` states that `resource(r)` cannot be duplicated. In particular, this means that the predicate can only be owned by one thread at a time. Note that in Raven the operator `&&` is interpreted as *separating conjunction*, which we denote by $*$ outside of code snippets. When a module implements an interface, all such axioms must be turned into lemmas that are supported by proofs. The tool simplifies this task by attempting to automatically complete the proofs of all axioms that have been omitted in a module implementation by discharging them directly to the SMT solver.

Thread modular reasoning is enabled via (shared) invariants like `lock_inv(l,r)` declared on §2 of interface `Lock`. Invariants specify *global* data structure-level logical invariants about shared resources such as the heap representation of the lock. They are themselves duplicable resources that can be freely shared between threads. For this to be sound, a thread can only access the underlying resources of an invariant for one atomic step at a time after which the invariant must be reestablished.

The procedure `create(r)` initializes a new ticket lock instance `l` and *binds* the resource `r` to `l` via the invariant `lock_inv(l,r)`. The invariant then governs the ownership transfer of `R.resource(r)` between threads that share `l`. The actual transfer is realized via

```
interface LockResource {
  rep type T
  // Resource protected by a lock
  pred resource(r: T)
  axiom exclusive(r: T)
    requires resource(r) && resource(r)
    ensures false
}

interface Lock {
  module R: LockResource

  inv lock_inv(l: Ref, r: R)

  proc create(r: R) returns (l: Ref)
    requires resource(r)
    ensures lock_inv(l, r)

  proc acquire(l: Ref, ghost r: R)
    requires lock_inv(l, r)
    ensures resource(r)

  proc release(l: Ref, ghost r: R)
    requires lock_inv(l, r) && resource(r)
    ensures true
}
```

(a) Lock interface

```
module TicketLock[R: LockResource] : Lock {
  field next: Int; field curr: Int

  module DisjInts = Library.DisjSet[IntType]
  module AuthDisjInts = Library.Auth[DisjInts]
  ghost field tkts: AuthDisjInts

  inv lock_inv(l: Ref, r: R) {
    exists n: Int, c: Int, b: Bool ::
      own(l.next, n) && n >= 0 && own(l.curr, c)
      && own(l.tkts, ●[0, n-1])
      && (b ? own(l.tkts, ○{c}) : resource(r))
  }

  proc acquire(l: Ref, ghost r: R)
    requires lock_inv(l, r)
    ensures resource(r)
  {
    ghost var lockAcq: Bool;
    unfold lock_inv(l, r)[lockAcq := b];
    val nxt := faa(l.next, 1);
    fpu(l.tkts, ●[0,nxt-1], ●([0, nxt], {nxt}));
    fold lock_inv(l, r)[b := lockAcq];
    var crr := -1;
    while (crr != nxt)
      invariant lock_inv(l, r) &&
        crr == nxt ? resource(r) : own(l.tkts, ○{nxt})
    {
      unfold lock_inv(l, r)[lockAcq := b];
      crr := l.curr;
      fold lock_inv(l,r)[b := crr == nxt || lockAcq];
    }
  }
  ...
}
```

(b) `TicketLock` module

Figure 1: Raven implementation of a ticket lock

the procedures `acquire` and `release`. For instance, the contract of `acquire` specifies that if lock `l` satisfies `loc_inv(l,r)` upon entry to the procedure, then `acquire` will guarantee access to `resource(r)` upon return. Because the invariant is duplicable, calls to `acquire` and `release` do not actually consume it, even though the contracts do not explicitly specify that ownership of the invariant is returned back to the calling context. Note that the parameter `r` is declared as **`ghost`** for these two procedures to indicate that it is only used for verification.

Implementation Language. Raven's underlying programming language is a first-order concurrent imperative language supporting primitive atomic statements such as fetch-and-add (**`faa`**), compare-and-set (**`cas`**), thread spawning, etc. Figure 1b shows a functor that implements the interface `Lock` using the ticket lock algorithm. You may ignore the code highlighted in gray for now in the implementation of `acquire` as it is only relevant for the proof, which we discuss later.

A ticket lock instance `l` maintains a logical FIFO queue of threads that are waiting to acquire the lock. The queue is represented by two counters stored in the heap fields

`next` and `curr`. The counter `next` tracks the next available ticket number. When a thread calls `acquire(l)` it reads and increments `l.next` in one atomic step using an **`faa`** operation (§2). The read integer value `nxt` represents the ticket number that determines the thread's place in the queue.

The counter `l.curr` always trails `l.next` and tracks the ticket number of the thread whose turn it is to acquire the lock and obtain ownership of `resource(r)`. After a thread has obtained its ticket number, it enters a loop that continuously reads `l.curr` until this counter has caught up to `nxt`, indicating that the thread has acquired the lock. The implementation of `release` (omitted in the figures) simply increments `l.curr`, signaling to the next thread in the queue that its turn has come.

Resources and Resource Algebras. The invariant `lock_inv(l, r)` for the ticket lock is defined on §2. In Raven, access to heap fields like `next` and `curr` is governed via fractional permission resources [7]. These provide a multi-reader, single-writer access control mechanism. In general, for a reference x, a heap field `f:T`, a value v of type `T`, and a rational $q \in (0, 1]$, the predicate $\mathbf{own}(x.\mathtt{f}, v, q)$ indicates that $x.\mathtt{f}$ stores value v and gives fractional access permission q to this location. A fraction $q > 0$ provides read access and $q = 1$ additionally gives write access to the location. If q is omitted, then it defaults to 1. The first three conjuncts of the invariant, thus, specify that the invariant provides the permission to read and write the locations `l.next` and `l.curr` and that the value `n` stored in `l.next` is positive.

The remaining two conjuncts of the invariant govern the ownership transfer of the protected resource using an auxiliary ghost field. Ghost fields allow users to augment the program state with logical *ghost resources* that track auxiliary information conducive to verification. Ghost resources a are drawn from a *resource algebra* (of which fractional permissions are one example). Formally, a resource algebra (RA) is a tuple:

$$\langle M,\ \mathcal{V}\colon M \to \mathbb{B},\ \varepsilon \in M,\ (\cdot)\colon M{\times}M \to M,\ (\backslash)\colon M{\times}M \to M,\ (\rightsquigarrow)\colon M{\times}M \to \mathbb{B}\rangle$$

that satisfies the axioms laid out in Fig. 2. In particular, $\langle M, (\cdot)\rangle$ is a commutative monoid with unit ε. For a reference x and ghost field g and value $a \in M$, the predicate $\mathbf{own}(x.g, a)$ states ownership of fragment a of the total ghost resource value stored at $x.g$. The monoid operation $(\cdot) : M \times M \to M$ provides meaning to separating conjunction of such owned fragments:

$$\mathbf{own}(x.g, a)\ *\ \mathbf{own}(x.g, b) \dashv\vdash \mathbf{own}(x.g, a \cdot b)\ .$$

The predicate $\mathcal{V}(a)$ states that the fragment a is *valid*. Only valid fragments can be owned: $\mathbf{own}(x.g, a) \vdash \mathcal{V}(a)$. The relation $a \rightsquigarrow b$ indicates that an owned fragment $\mathbf{own}(x.\mathtt{g},\ a)$ can be updated to $\mathbf{own}(x.\mathtt{g},\ b)$ in a frame-preserving way, i.e., without invalidating ownership of other fragments $\mathbf{own}(x.\mathtt{g},\ c)$ of the same ghost location (FPU-VALID). The *subtraction* function $(\backslash)$ can be understood as the right inverse of $(\cdot)$. Its inclusion in the definition of RAs deviates from Iris's notion of an RA. It is used for automating the *frame rule* of separation logic using an SMT solver. We discuss this in more detail in §3.

An example of a resource algebra that we use in the invariant of the ticket lock is $\mathsf{DisjSet}[T]$, which consists of subsets $S \in \wp(T)$ of some carrier set T with disjoint set

$$\mathcal{V}(\varepsilon) \qquad \text{(ID-VALID)}$$
$$\forall a, b, c :: (a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{(COMP-ASSOC)}$$
$$\forall a, b :: a \cdot b = b \cdot a \qquad \text{(COMP-COMM)}$$
$$\forall a :: a \cdot \varepsilon = a \qquad \text{(COMP-ID)}$$
$$\forall a, b :: \mathcal{V}(a \cdot b) \implies \mathcal{V}(a) \wedge \mathcal{V}(b) \qquad \text{(COMP-VALID)}$$
$$\forall a :: a \setminus \varepsilon = a \qquad \text{(FRAME-ID)}$$
$$\forall a, b :: \mathcal{V}(a \setminus b) \implies (a \setminus b) \cdot b = a \qquad \text{(COMP-FRAME-INV)}$$
$$\forall a, b :: \mathcal{V}(a \cdot b) \implies \mathcal{V}((a \cdot b) \setminus b) \qquad \text{(FRAME-COMP-VALID)}$$
$$\forall a, b, c :: a \rightsquigarrow b \wedge \mathcal{V}(a \cdot c) \implies \mathcal{V}(b \cdot c) \qquad \text{(FPU-VALID)}$$

Figure 2: Axioms defining a resource algebra

union as composition:

$$\mathsf{DisjSet}[T] ::= S \mid \lightning \quad \mathcal{V}(a) = (a \neq \lightning) \quad a \cdot b = (\mathcal{V}(a) \wedge \mathcal{V}(b) \wedge a \cap b = \varnothing)\,?\,a \cup b : \lightning$$
$$\varepsilon = \varnothing \qquad a \setminus b = (\mathcal{V}(a) \wedge \mathcal{V}(b) \wedge b \subseteq a)\,?\,a - b : \lightning \qquad a \rightsquigarrow b = \mathit{false}\ .$$

Raven allows users to define their own proof-specific RAs using modules that implement the predefined interface `Library.ResourceAlgebra`. This interface defines the signature of the RA operations along with their axioms. When specifying a new RA, it often suffices to define only the operations, and Raven automatically verifies that the axioms are satisfied. To further simplify the definition of new RAs, the standard library defines specialized variants *cancellative* RAs and *lattice-based* RAs, which satisfy stronger properties.

As certain reasoning patterns often recur across proofs, Raven ships with several predefined RAs as well as RA functors that construct new RAs from existing ones. In fact, $\mathsf{DisjSet}[T]$ is predefined as `Library.DisjSet[T]`.

The second common RA construction that we use in the invariant is that of the *authoritative RA* [22]. This construction is useful whenever we want to track an updatable ghost resource a in an invariant, but allow fragments of a to be owned by individual threads. Formally, given an RA M, the authoritative RA $\mathsf{Auth}[M]$ is defined as:

$$\mathsf{Auth}[M] ::= \bullet(a, b) \mid \circ a \mid \lightning \qquad \varepsilon = \circ M.\varepsilon$$
$$\bullet(a, b) \cdot \circ c = \circ c \cdot \bullet(a, b) = \bullet(a, b \cdot c) \qquad \circ a \cdot \circ b = \circ(a \cdot b) \qquad \_ \cdot \_ = \lightning \text{ otherwise}$$
$$\mathcal{V}(\bullet(a, b)) = \mathcal{V}(a) \wedge \mathcal{V}(b) \wedge \mathcal{V}(a \setminus b) \qquad \mathcal{V}(\circ a) = \mathcal{V}(a) \qquad \mathcal{V}(\lightning) = \mathit{false}$$
$$\bullet(a_1, b_1) \rightsquigarrow \bullet(a_2, b_2) \;=\; \mathcal{V}(\bullet(a_2, b_2)) \wedge (\forall c :: a_1 = b_1 \cdot c \Rightarrow a_2 = b_2 \cdot c)$$

where $a, b, c \in M$. We omit the definition of $(\setminus)$. Intuitively, $\bullet(a, b)$ denotes the authoritative resource a with a fragment b of a that can still be handed out. On the other hand, $\circ c$ denotes a fragment c of a that has already been handed out. We write just $\bullet a$ for $\bullet(a, M.\varepsilon)$. The authoritative RA is predefined as `Library.Auth[M]`.

The invariant `lock_inv(l,r)` uses the ghost field `tkts` of type $\mathsf{Auth}[\mathsf{DisjSet}[\mathbb{Z}]]$ (lines 30-32) to track the authoritative set of tickets `[0,n-1]` that have been in use so

far (§2). When a thread attempts to acquire the lock and increments `next` to `n+1`, the authoritative set of tickets is increased to include `n`. The fragment `○{n}` is given to the thread as proof that it exclusively owns ticket `n` while it is waiting for its turn to acquire the lock. Once the value `c` of `curr` has caught up to `n`, the thread can exchange `○{n}` for `resource(r)`. This ownership transfer is realized via the conjunct on §2. That is, the Boolean `b` indicates whether the lock is presently held by a thread. When the thread releases the lock and increments `curr`, then §2 forces the thread to relinquish ownership of `resource(r)` back to the invariant because the thread does not also own `○{c+1}`.

Raven as an IVL. Raven's module system and user-definable ghost state provide the flexibility to model a wide range of front-end language features, thus increasing its usefulness as an IVL. The module system facilitates rapid prototyping of various features, while custom resource algebras can be used to encode language features that are not directly supported by Raven. As a concrete example, user-definable ghost resources can be used to encode deallocation obligations for reasoning about languages with manual memory management [5]. Additionally, Raven includes explicit `inhale` and `exhale` commands for direct manipulation of the proof state. Raven also supports atomic triples, which are discussed in §4. Invariants and atomic triples can be used to define new atomic primitives. When developing frontend verifiers, one only needs to reason about the encoding of the concurrency primitives (e.g. that they satisfy their atomic specifications), but not the soundness of the entire underlying concurrent program logic.

3 Proof Automation

Proofs in Raven deploy a combination of user-provided ghost code annotations and SMT-based proof automation. We demonstrate the mechanics using the proof of `acquire` (Fig. 1b with ghost code highlighted in gray). We first discuss the role of the annotations and then explain the SMT encoding that enables proof automation.

Proof Annotations. As discussed earlier, a thread executing `acquire(l,r)` first reads and increments `l.next` using an **`faa`**. This statement requires read and write access to `l.next`, which is granted by §2 of the invariant `lock_inv(l, r)`. In order to retrieve the relevant permission from the invariant, the proof author needs to manually unfold the invariant, which is achieved by the **`unfold`** statement on §2. The statement instructs the verifier to exchange the symbolic resource denoting ownership over the invariant, with the resources contained in the *body* of the invariant. The statement also assigns the value of the existentially quantified `b` in the body of the invariant to the ghost variable `lockAcq` so that we can subsequently refer to it in the proof.

As the invariant is shared between all threads that have access to `l`, other threads may interfere between each atomic access to `l.next`. The verifier therefore enforces that the invariant is *closed* again between any two such atomic steps. This is achieved using the **`fold`** statement as used on §2. The verifier is able to automatically compute witnesses for the existentially quantified variables `n` and `c` in the invariant, as these can be inferred from the heap state. However, the value of `b` needs to be supplied manually. Here, we set it to `lockAcq` since the current thread has not yet acquired the lock. When the verifier executes the **`fold`** statement, it first checks that the invariant indeed holds again for the

computed and supplied existential witnesses. Then it exchanges the related resources with the symbolic resource representing the closed invariant.

Most of the remaining ghost code is similarly related to unfolding and folding the invariant around atomic accesses to the underlying resources. The one exception is §2, which we discuss in more detail.

Suppose that after the value of `l.next` has been incremented to `nxt+1` by the **`faa`**, we immediately attempted to fold the invariant. Then the **`fold`** statement would fail because the invariant is no longer satisfied: the ghost location `l.tkts` still holds `•[0, nxt-1]` but needs to hold `•[0, nxt]`. To bring `l.tkts` back into sync with the value stored at `l.next`, §2 performs a frame-preserving update (**`fpu`**) that replaces the fragment `•[0, nxt-1]` of `l.tkts` with `•([0, nxt], {nxt})`. The updated fragment can be split into the fragments `•[0, nxt]` and `○{nxt}`. The first part is folded into the invariant on §2. The second part stays with the thread and is the proof of ownership of ticket `nxt` that is later traded for `resource(r)` with the invariant (§2 if `crr == nxt`).

SMT Encoding. Raven automatically checks the proof outline by generating a verification condition that is discharged using an SMT solver.

Raven's first-order logic encoding maintains for each field f a *field heap* f`#heap`, which maps references ℓ to values of the resource algebra associated with f. The value f`#heap[`ℓ`]` represents the total fragment of the location $\ell.f$ that is presently owned by the thread. Heap fields and ghost fields are treated uniformly: the RA associated with a heap field is that of fractional permissions over the values of the field's underlying type.

Similarly, the encoding maintains *invariant heaps* and *predicate heaps* that track all instances of invariants and predicates owned by the thread. The representation of these heaps is similar to that of field heaps but relies on specialized RAs. In particular, for an invariant $inv(\overline{x})$, the invariant heap inv`#heap` maps a valuation $\overline{v}$ of the parameters $\overline{x}$ to a Boolean flag that indicates whether $inv(\overline{v})$ is owned by the current thread.

Ghost statements like **`fold`**, **`unfold`**, and **`fpu`** are then translated to updates of the various heaps to reflect the changes to the owned resources affected by the execution of the statement. The encoding maintains the property that all heaps are valid with respect to the underlying RA. Note that unfolding and folding an invariant does not change the invariant heap since invariants are duplicable resources. Instead, a separate atomicity analysis guarantees that the same invariant cannot be unfolded twice in a row without folding it in between.

To build intuition for the reduction to SMT, we discuss the encoding of the frame-preserving update on §2 in more detail:

```
/* Encoding of fpu(l.tkts, •[0, nxt-1], •([0, nxt], {nxt})); */
// 1. Check that fpu is valid
assert •[0, nxt-1] ⇝ •([0, nxt], {nxt});
// 2. Exhale own(l.tkts, •[0, nxt-1])
tkts#heap := tkts#heap[l := (tkts#heap[l] \ •[0, nxt-1])];
assert 𝒱(tkts#heap[l]);
// 3. Inhale own(l.tkts, •([0, nxt], {nxt}))
tkts#heap := tkts#heap[l := (tkts#heap[l] · (•[1, nxt], ○{nxt}))];
```

Since the **`fpu`** updates `l.tkts`, only the field heap `tkts#heap` is affected. §3 first checks that the update is indeed frame-preserving by utilizing the $(\rightsquigarrow)$ relation of the underly-

ing RA, Auth[DisjSet[$\mathbb{Z}$]]. The actual update then proceeds in two steps, first we *exhale* ownership of the fragment `•[0, nxt-1]` by removing it from the field heap at `l` (§3). This step uses the function $(\backslash)$ to compute the residual fragment that is still owned by the thread after the removal of `•[0, nxt-1]`. Because the **`fpu`** may be executed in a state where the thread does not actually own `•[0, nxt-1]`, §3 asserts that the resulting heap remains valid. Together with the RA axiom COMP-FRAME-INV, this check also ensures that the newly computed fragment `tkts#heap[l]` is indeed a residual of `•[0, nxt-1]` and the old `tkts#heap[l]`. That is, **`own`**`(l.tkts,•[0, nxt-1])` actually holds before the update.

The second step composes the new fragment (`•[1, nxt], ○{nxt}`) of the **`fpu`** to the computed residual (§3). The axiom FPU-VALID and the check on §3 guarantee that the resulting field heap is again valid. The **`fpu`** involves an implicit application of the frame rule in separation logic with the residual fragment acting as the frame.

The SMT reduction then utilizes a standard SSA encoding of this intermediate representation of the program, using the theory of arrays to reason about the updates to heap fields. To automate reasoning about RAs, the axioms from Fig. 2 are annotated with appropriate E-matching patterns. If a module abstracts over a generic RA, then during the verification of the module, the axioms are given directly to the SMT solver. For concrete user-defined instantiations of RAs, the tool provides the SMT solver with axioms that define the RA operations as specified by the user, as well as the RA axioms (except for COMP-ASSOC). Providing the RA axioms for user-defined RAs is usually redundant from a completeness perspective but we find that it improves solver performance. The associativity axiom is omitted because of its cubic instantiation cost in the number of RA terms. (That is, we rely on the SMT solver being able to infer associativity from the definitional axioms of the RA operations.)

With this encoding, Raven verifies the full implementation of the ticket lock in less than one second. We emphasize that Fig. 1b comprises all annotations that the user must provide for the proof of `acquire`. The annotation burden for `release` is similar.

4 Additional Features

In this section, we discuss several additional features of Raven that aim to further increase its expressivity and usability.

Atomic Procedure Contracts. A common usage of concurrent program logics is to prove linearizability of concurrent data structures. Logics like Iris provide the notion of an *atomic triple* [43] to enable compositional reasoning about linearizability. While atomic triples can be encoded using standard Hoare triples and invariants, Raven supports them directly via atomic procedure contracts.

An atomic triple takes the form $\langle\, \overline{x}.\, P \,\rangle\ st\ \langle\, \overline{r}.\, Q \,\rangle$. Intuitively, the atomic precondition P acts like an invariant up to the linearization point of statement st. That is, before the linearization point, st has access to the resources provided by P, but it needs to ensure that P is maintained by each of its atomic steps. The logical variables $\overline{x}$ should be thought of as representing the abstract state of the data structure that st operates on. Importantly, between any two steps of st the values of $\overline{x}$ for which P holds may change due to interferences by other threads. At its linearization point, st must transform P into Q in one atomic step. The variables $\overline{r}$ are the values that st returns upon termination.

```
pred is_lock(l: Ref; r: R, b: Bool) {
  exists n: Int, c: Int ::
    own(l.next, n) && n >= 0
    && own(l.curr, c)
    && (b ? own(l.tkts, ○{c}) : resource(r))
    && own(l.tkts, ●[0, n-1])
}

proc wait_loop(l: Ref, x: Int,
    implicit ghost r: R,
    implicit ghost b: Bool
)
  requires own(l.tkts, ○{x})
  atomic requires is_lock(l, r, b)
  atomic ensures is_lock(l, r, true)
      && b == false && resource(r)
{
  ghost val phi := bindAU();
  r, b := openAU(phi);
  unfold is_lock(l);
  val c: Int := l.curr;

  if (x == c) {
    fold is_lock(l, r, true);
    commitAU(phi);
    return;
  } else {
    fold is_lock(l, r, b);
    abortAU(phi);

    r, b := openAU(phi);
    wait_loop(l, x);
    commitAU(phi);
  }
}
```

```
proc acquire(l: Ref,
    implicit ghost r: R,
    implicit ghost b: Bool)
  atomic requires is_lock(l, r, b)
  atomic ensures is_lock(l, r, true)
      && b == false && resource(r)
{
  ghost val phi := bindAU();

  r, b := openAU(phi);
  unfold is_lock(l);
  val nxt: Int := l.next;
  fold is_lock(l, r, b);
  abortAU(phi);

  r, b := openAU(phi);
  unfold is_lock(l);
  val res: Bool := cas(l.next, nxt, nxt+1);

  if (res) {
    fpu(l.tkts, ●[0,nxt-1], ●([0, nxt], {nxt}));
    fold is_lock(l, r, b);
    abortAU(phi);

    r, b := openAU(phi);
    wait_loop(l, nxt);
    commitAU(phi);
  } else {
    fold is_lock(l, r, b);
    abortAU(phi);
    r, b := openAU(phi);
    acquire(l);
    commitAU(phi);
  }
}
```

Figure 3: Atomic triple specification for `TicketLock.acquire`

After the linearization point, the transformed resources Q are then no longer accessible by st. That is, the atomic triple captures the effect of st on the underlying data structure at its linearization point in relationship to its return values.

We explain Raven's support for atomic triples using the variant of the ticket lock example shown in Fig. 3. An atomic triple for a procedure is specified by marking its requires and ensures clauses as **`atomic`**. Such atomic pre/postconditions can be mixed with regular pre/postconditions (as on §4), which retain their usual semantics. The variables $\overline{x}$ of the atomic triple are specified as *implicit ghost* parameters in the atomic contract. Note that unlike the invariant-based specifications used in Fig. 1a, the atomic contract of `acquire` now allows us to directly refer to the state `b` of the lock. It specifies that, at `acquire`'s linearization point, the state of the lock changes from `b == false` (unlocked), to `b == true` (locked). To demonstrate the interplay between atomic contracts of different procedures, we split the loop in the original implementation of `acquire` into a separate recursive procedure `wait_loop`.

Raven's atomic contracts work as follows. Each call to a procedure with an atomic contract has an associated ghost resource, the atomic update, which tracks the state of the atomic contract throughout the procedure's execution. These resources are uniquely identified by *atomic tokens*. The statement **bindAU** can be used to obtain a handle on the atomic token associated with the call to the current procedure (e.g. §4). Using this handle, ghost code can then manipulate the state of the atomic update. For example, if code needs access to a resource in the atomic precondition of the update, then this is achieved with the statement **openAU** (e.g. §4). The statement returns the current values of the logical variables $\overline{x}$ associated with the atomic update. After opening the atomic update and taking an atomic step, it needs to be closed again. This can be done in two ways. The statement **abortAU** (e.g. §4) checks that the atomic precondition P still holds for the same values $\overline{x}$ that were obtained when opening the atomic update and then relinquishes ownership of the associated resources back to the atomic update. In contrast, the statement **commitAU** is used at the linearization point to check that Q holds for the provided return values (e.g. §4). It then transfers ownership of Q to the atomic update and marks it as committed. Raven checks at each return point of an atomic procedure, that the associated atomic update has indeed been previously committed for the actual return values.

Implicit Predicate Parameters. Predicates and invariants also support implicit parameters. These are separated from the input parameters with a semicolon in the predicate definition, such as `r:R` and `b:Bool` in §4 of Fig. 3. This indicates that if a thread owns `is_lock` for some `l`, then the values of `r` and `b` can be inferred from the symbolic state of that predicate instance. For example, this allows us to omit these arguments when unfolding `is_lock(l)`, since Raven can automatically compute the appropriate witness for the resulting existential quantifiers. Raven's witness computation feature is discussed in more detail below. The ghost resource associated with a predicate `p(`$\overline{x}$`;` $\overline{y}$`)` tracks exactly one valuation of the implicit parameters $\overline{y}$ for each valuation of the parameters $\overline{x}$. To guarantee soundness, Raven checks that it is impossible to *simultaneously* have ownership of the resources contained in `p(`$\overline{x}$`;` $\overline{y_1}$`)` and `p(`$\overline{x}$`;` $\overline{y_2}$`)` for any $\overline{y_1} \neq \overline{y_2}$.

Iterated Separating Conjunctions. Reasoning about unbounded memory regions in separation logic, e.g. to express ownership over array segments, requires support for *iterated separating conjunctions* (ISCs). Raven supports automated reasoning about ISCs using an SMT encoding that adapts the technique proposed in [36]. We extend this technique by generalizing it to resources over arbitrary RAs and by providing heuristic support for dealing with $\forall\exists$ quantifier alternations.

For illustrative purposes, we present a simplified version of a predicate that we use in the Raven implementation of a case study discussed in more detail in §5.2:

```
pred cssR(r: Ref) {
  forall n: Ref :: n in nodeSet(r) ==>
    exists b: Bool, cn: Set[K] ::
      own(n.lock, b, 1.0) && (b ? true : nodePred(r, n, cn))
}
```

The predicate `cssR(r: Ref; nodeSet: Set[Ref])` expresses ownership of the resource `nodePred(r, n, cn)` associated with each node `n` in the set `nodeSet(r)`, provided `m`'s `lock` bit is `false` (i.e., the node is unlocked). Intuitively, the nodes constitute a data structure

associated with the given root node `r`. The existentially quantified `cn` is a set of keys that represents the logical state of node `n`.

Raven handles such assertions as follows. When an ISC is assumed to hold (e.g., if it occurs in a precondition), any nested existential quantifiers are skolemized. The remaining universal quantifiers are translated to universally quantified first-order formulas, which are sent to the SMT solver. Following [36], the supported ISCs must adhere to certain restrictions so that the quantifiers can be automatically annotated with appropriate triggers to ensure robust proof automation using E-matching.

When an ISC is asserted to hold, the outermost universal quantifier is skolemized and the inner existential quantifier turns into a universal quantifier. Rather than passing such quantifiers on to the SMT solver directly, Raven provides heuristic support for instantiating them upfront by computing appropriate witness terms, say `b_wtns(n)` and `cn_wtns(n)` in the example above.

Existential Witness Computation. The witness computation heuristic exploits the fact that the values of existential quantifiers in separation logic formulas are often uniquely determined by the underlying (ghost) heap state. In our example, the value of `b` is determined by `n`'s `lock` location and the value of `cn` is determined by `nodePred` whose third parameter is implicit. This allows Raven to compute the following constraints on the witness terms, expressed in terms of the field and predicate heaps used for the SMT encoding of the resources:

```
n in nodeSet ==> b_wtns(n) == lock#Heap(n)#0
n in nodeSet && !b ==> cn_wtns(n) == (node#PredHeap[ (r, n) ])
```

In this example, the witness computation and ISC encoding together enables automated reasoning about the predicate `cssR` without requiring the proof author to manually instantiate universal quantifiers or provide witness terms.

5 Implementation and Evaluation

Raven is implemented in about 16K lines of OCaml code. The tool operates on source programs written in the Raven language. We additionally provide integration into Visual Studio Code via a rudimentary language server written in TypeScript. The current test and benchmark suite consists of over 9K lines of Raven code with the largest case study (discussed in §5.2) in the order of 1K lines (including specifications and proofs). The tool and benchmarks are made available with the artifact accompanying this paper [16].

After parsing and type checking, the input program is compiled into verification conditions expressed in SMT-LIB format. These are then dispatched with Z3 [33], utilizing the solver's incremental solving capability.

Evaluation. We evaluate Raven's effectiveness as a verification tool by comparing it with existing automation efforts for Iris. We focus our comparison on Diaframe [34,35], a proof search engine built for the Rocq mechanization of Iris that helps to automate linearizability proofs for fine-grained concurrent data structures. While there exist a number of other tools that automate proofs in concurrent separation logics, these tools make different trade-offs in terms of generality vs. degree of automation compared to

Raven. Moreover, none of these tools are based directly on Iris. We therefore focus on Diaframe for our experimental comparison and discuss other tools in §6.

In our first experiment (§5.1), we reimplement Diaframe's benchmark suite of data structures in Raven and compare the performance of the two tools on these benchmarks. While Diaframe is able to automatically construct foundational proofs of correctness when it works, its approach to automation comes with certain tradeoffs, particularly when dealing with proof-specific resource algebras, and when scaling to more complex developments. In order to examine how the tools perform in such cases, our second experiment (§5.2) reimplements the proof of one of the concurrent *template* algorithms of [27] (which was originally done in Iris) and its instantiation to B+ trees (which was originally done using the SMT-based verifier GRASShopper [42]).

To compare Raven's performance with other SMT-based tools, we conduct an experiment (§5.3) that compares Raven with Viper on examples that are supported by both tools. Finally, we conduct an additional experiment (§5.4) where we inject bugs in Raven programs to determine performance for failing programs.

5.1 Experiment 1: Comparison with Diaframe on Diaframe benchmarks

Diaframe implements a goal-directed proof search engine using ideas from linear logic programming and biabduction. It relies on a large library of hints, as well as custom user-provided hints to guide the proof search. Foundational tools like Rocq provide stronger correctness guarantees than SMT-based tools as they rely on a much smaller trusted computing base. However, they typically require a user to spell out proofs in much more detail. Diaframe aims to automate a lot of this low-level proof burden while still ensuring foundational correctness.

We reimplemented Diaframe's benchmark suite of 23 examples in Raven[3]. It consists of 19 concurrent data structures including different variants of locks, counters, sets, queues, and stacks, as well as four clients for some of these data structures. We additionally include the implementation of a *fractional token* resource algebra that is used in several of the benchmark examples.

We aimed to stay as close as possible to Diaframe's implementations and specifications. Some differences in the specifications arise due to Raven's more restricted support of higher-order features. For example, our lock implementations parameterize over the locked resources using abstract predicates and higher-order modules (as in Fig. 1a). In contrast, the Diaframe versions directly use higher-order quantification over Iris assertions. This makes the Diaframe specifications slightly more general than the Raven versions. So far we have not encountered situations where the more restrictive specification was a hindrance to verification of clients. However, one can easily construct artificial examples that cannot be verified using the Raven specifications.

The results of our comparison are summarized in Table 1. We list the size of each benchmark (measured in the number of program instructions and declarations in the Raven implementation). In addition, for each tool we measure the number of proof-related declarations "pf decl" (e.g., ghost fields, RA functor instantiations, Iris name

[3] We merge Diaframe's 'lclist' and 'lclist-extra' benchmarks, resulting in 23 examples instead of 24 in the original paper.

	Raven					Diaframe		
benchmark	**size**	**pf decl**	**pf instr**	**pf ovrhd**	**runtime**	**pf decl**	**pf instr**	**runtime**
arc	24	18	50	*2.83*x	0.36 s	15	0	9.48 s
bag stack (aka Treiber stack)	20	6	32	*1.90*x	0.40 s	24	19	16.89 s
barrier	44	31	90	*2.75*x	3.90 s	35	6	379.49 s
bounded counter	13	4	7	*0.84*x	0.16 s	18	6	12.59 s
cas counter	12	11	12	*1.91*x	0.20 s	19	0	8.51 s
clh lock	25	11	13	*0.96*x	0.34 s	26	0	18.84 s
fork join	11	11	7	*1.63*x	0.19 s	19	0	7.45 s
inc dec	16	4	16	*1.25*x	0.18 s	22	0	22.64 s
lclist	114	25	46	*0.62*x	0.63 s	51	21	126.57 s
mcs lock	32	16	36	*1.62*x	1.24 s	31	0	50.75 s
msc queue	34	13	29	*1.23*x	0.34 s	16	0	88.83 s
peterson	29	11	40	*1.75*x	1.23 s	32	26	-
queue	36	14	32	*1.27*x	0.25 s	20	0	49.46 s
spin lock	10	6	8	*1.40*x	0.20 s	23	0	7.22 s
rwlock duolock	45	14	23	*0.82*x	0.45 s	22	0	16.43 s
rwlock lockless faa	19	7	25	*1.68*x	0.37 s	18	0	21.10 s
rwlock ticket bounded	30	22	39	*2.03*x	0.83 s	30	3	39.00 s
rwlock ticket unbounded	31	14	38	*1.67*x	0.50 s	30	0	17.50 s
ticket lock	16	17	12	*1.81*x	0.72 s	28	0	19.80 s
barrier client	35	50	87	*3.91*x	0.74 s	32	24	-
cas counter client	12	6	4	*0.83*x	0.20 s	10	0	5.68 s
fork join client	10	6	3	*0.90*x	0.18 s	9	0	3.76 s
ticket lock client	15	5	6	*0.73*x	0.21 s	11	0	5.61 s
tokens ra	0	54	46	-	0.34 s	131	290	18.05 s
Average				*1.65*x	0.54 s			26.96 s

Table 1: Comparison of Raven and Diaframe on Diaframe's benchmark suite; runtimes averaged over 10 runs. size = number of program instructions; pf decl = number of proof-related declarations; pf instr = number of proof instructions; pf ovrhd = proof overhead defined as (pf decl+pf instr)/size.

space declarations), proof-related instructions "pf instr" (e.g., ghost commands, proofs and invocations of auxiliary lemmas, proof tactic invocations, Diaframe hints), the proof overhead "pf ovrhd" defined as the ratio between "size" and "pf decl"+"pf instr", and the tool's runtime for the verification. Runtimes are measured on an Apple M1 Pro (32 GB RAM, 10 cores) and are averaged over 10 runs. We are unable to report runtimes for "peterson" and "barrier client" due to compilation issues on Diaframe's master branch.

We find that Raven requires considerably fewer proof declarations than Diaframe. This can be attributed to Raven's restricted support of higher order features, which reduces the amount of required boilerplate code. Proof overhead for Raven implementations varies between 0.62x to 3.91x, with a mean of 1.65x.

As an illustrative example, if we examine the 'barrier client' benchmark closely we find that out of its 87 proof instructions, 46 instructions pertain to (un-)folding predicates and 23 pertain to (un-)folding invariants, while the remaining 18 instructions

relate to lemma calls, frame-preserving updates, etc. This is typical for the other benchmarks as well. Certain heuristics can be applied automatically to infer some of these instructions, however it may lead to slowdowns and inconsistent behaviour for the user. The development of robust heuristics for further proof automation is a lucrative future direction of research.

Owing to its considerable hints library, Diaframe is able to construct many of these proofs in their entirety. However, the automation comes at the cost of flexibility. When the user wants to use a resource algebra that is not supported by Diaframe out of the box, they are required to supply such hints manually. This is reflected in the 'tokens ra' benchmark, which contains the definition of the *fractional tokens* resource algebra that is used in the proofs of several of the data structures including "barrier", "rwlock duolock", "arc", etc. While Raven's implementation required 54 proof declarations and 46 proof instructions, Diaframe required 131 and 290, respectively.

These hints, broadly speaking, serve a similar purpose as Raven's RA axioms for the subtraction operator $(\backslash)$. Raven provides SMT-based automation for proving that these axioms are satisfied for a particular RA definition, while Diaframe requires users to define these hints directly in Rocq/Iris with little automation.

Comparing runtimes, we find that Raven is typically between one to two *orders of magnitude* faster. While Diaframe has to perform the much harder task of searching through the space of proof tactics, Raven directly encodes separation logic reasoning to first-order logic and dispatches it to Z3.

In our experience, Raven's flexibility in adapting to user-defined resource algebras, as well as significantly faster runtimes make a considerable difference during the process of developing correctness proofs, by enabling rapid prototyping and iteration.

5.2 Experiment 2: The GIVEUP Template Case Study

The goal of our second experiment is to evaluate the performance of Raven on more complex verification tasks. For this purpose, we chose to reimplement the proof of one of the algorithms for concurrent search structures from [27] in Raven. Specifically, we reimplemented the *give-up* algorithm.

The original proof consists of two parts. The first part is a proof of linearizability of a template algorithm for a concurrent set data structure. It was mechanized using Iris's Rocq proof mode without the help of Diaframe. The template proof abstracts from the memory representation of the data structure by assuming *helper functions* that perform the node-local operations like inserting a key into a node. This way, the linearizability proof can be instantiated for vastly different concrete concurrent set implementations. The second part of the proof concerns the verification of the concrete helper function implementations, which only involves sequential reasoning. This part of the proof was originally done in the SMT-based verifier GRASShopper [42] for two concrete data structures: B+ trees and hash tables. We focus on the B+ tree instantiation.

The case study is a compelling target for our experiment because it heavily exercises Raven's higher-order module system and support for ISCs. The proofs also use complex proof-specific RAs (e.g., *keysets* [27] and *flows* [28,29]) to achieve the desired parametricity in the low-level memory representation of the data structure. Finally, the

	Raven				Iris			GRASShopper		
component	**size**	**pf decl**	**pf instr**	**runtime**	**pf decl**	**pf instr**	**runtime**	**pf decl**	**pf instr**	**runtime**
ccm	0	25	4	$0.13s$	107	484	$2.13s$	14	5	$0.12s$
flows-ra	0	37	22	$0.75s$	83	1804	$23.90s$	32	258	$6.03s$
keyset-ra	0	23	0	$0.26s$	27	661	$24.55s$	-	-	-
give-up	38	57	120	$7.60s$	56	465	$21.13s$	-	-	-
b-plus-tree	47	33	42	$2.10s$	-	-	-	18	24	$10.60s$
array-utils	57	75	60	$3.13s$	-	-	-	21	51	$10.18s$

Table 2: Comparison of the GIVEUP template implementation in Raven vs Iris + GRASShopper; runtimes averaged over 10 runs. size = number of program instructions; pf decl = number of proof-related declarations; pf instr = number of proof instructions.

fact that the original proof used both Iris/Rocq and an SMT-based tool makes for an interesting comparison point.

Table 2 shows the comparison of proof effort and verification times for the old and new proofs of the case study, aggregated according to the top-level components of the implementation and proof. As can be observed, Raven significantly reduces the proof effort and verification time compared with the Iris/Rocq mechanization. While the components mechanized in Iris amounted to a total of 273 proof declarations and 3414 proof instructions, the same components in Raven added up to 142 proof declarations and 146 proof instructions, while still being between 5x and 90x faster on each individual component. For the sequential components, Raven's performance is at par with GRASShopper; Raven ended up with 170 proof declarations and 128 proof instructions as opposed to GRASShopper's 85 and 338 respectively. This is to be expected since both tools depend on an SMT backend and provide similar levels of automation for reasoning about sequential code. Raven also provides faster runtimes than GRASShopper. This can be attributed to the fact that GRASShopper deploys its own E-matching engine as a preprocessing step to the SMT solver in order to provide completeness guarantees for certain decidable SL fragments [40, 41].

Diaframe Comparison. For the purpose of comparing Raven with Diaframe on a complex verification task, we attempted reimplementing the give-up proof using Diaframe. In our (anecdotal) experience, Diaframe struggles with verification of recursive functions, requiring manual user input around proof steps that involve inductive reasoning. This resulted in us having to guess *magic* parameters such as the maximal number of steps to attempt in the proof search so that the inductive hypothesis can be applied correctly. Similar difficulties arise when Diaframe fails to apply user-provided hints automatically and the user needs to carefully guide the proof search.

To be able to use Diaframe to its fullest capacity, the user needs to have a deep understanding of the Rocq formalization of Iris. Diaframe generates side conditions from proof steps in the search strategy. The authors, who are familiar with Iris, report instances where the origin or the proof of such side conditions was not clear.

The proof of give-up relies on complex resource algebras (flows and keysets) as mentioned earlier. Diaframe provides no support for proving that flows and keysets form a resource algebra. In addition, the proofs require elaborate custom hints to infer facts

that follow from the algebraic structure of the resource algebras. The overall result is an increased burden on the user.

In summary, we believe that at least for now, significant automation gains by using Diaframe do not materialize for large proof developments due to the above reasons. The unpredictability of the proof search can lead to a brittle user experience, in particular during development when the program and its specification is still in flux. Finally, Diaframe's lack of sufficient support for custom resource algebras may lead to users having to fall back on vanilla Iris. However, we do note that Diaframe can provide impressive automation gains on smaller examples and is designed to support full Iris rather than a restrictive fragment. Moreover, unlike Raven, it constructs proof objects in Iris/Rocq that provide foundational correctness guarantees.

5.3 Experiment 3: Comparison between Raven and Viper

To determine how Raven compares with other SMT-based separation logic verifiers, we conduct an experiment where we reimplement a subset of Viper's example set[4] in Raven and compare the runtimes. We compare Raven with Viper's verification-condition generation backend Carbon, as well as its symbolic execution backend Silicon.

Our results are summarized in Table 3a. The column 'size' refers to the number of program instructions in the Raven implementation. Examples marked with (*) are faulty examples which yield verification failure on all three tools. We modified the original proof of the 'tree delete' benchmark to avoid the use of magic wand. We also prove only memory safety for this example since Raven does not support sequence types. We translated the modified version of this benchmark back to Viper to obtain a fair comparison.

We note that Raven is using similar techniques as Viper's Carbon backend, but translates directly to SMT rather than going through Boogie. We believe that the difference in runtimes in cases where Raven outperforms Viper are likely attributed at least in part to the Java Virtual Machine (and Common Language Runtime) startup time. So the runtimes will likely be more closely matched in a practical scenario where Viper is used in a language server mode.

5.4 Experiment 4: Comparison of successful vs failing Raven examples

We also inject bugs in a subset of our benchmarks to assess Raven's performance on failing examples and give a more well-rounded picture. Our results are summarized in Table 3b. We find most failing runtimes to be comparable to the succeeding runtimes.

6 Related Work

Raven has been designed as an intermediate verification language and backend (IVL) that aims to serve (deductive) verification tools targeting concurrent programs. Its basic philosophy follows that of other IVLs like Why3 [13], Boogie [3], and Viper [37] in

[4] http://viper.ethz.ch/examples/

benchmark	size	Raven	Viper (Carbon)	Viper (Silicon)
adt*	3	0.1 s	2.1 s	3.4 s
array max	12	0.3 s	2.4 s	3.0 s
binary search	13	0.2 s	2.0 s	2.4 s
dutch flag	22	0.3 s	2.2 s	3.0 s
graph marking*	33	10.3 s	2.6 s	3.7 s
tree delete	23	0.5 s	4.5 s	3.6 s

(a) Comparison of Raven with Viper's backends Silicon and Carbon on a subset of Viper's examples.

benchmark	valid	buggy
adt	0.1 s	0.1 s
arc	0.4 s	0.3 s
barrier	7.5 s	12.6 s
graph marking	0.3 s	10.3 s
lclist	0.8 s	0.5 s
peterson	1.7 s	0.7 s
rwlock duolock	0.5 s	0.4 s

(b) Comparing Raven runtimes on valid vs. buggy benchmarks.

Table 3: Comparison of Raven with Viper and on faulty/buggy benchmarks.

that it automates a Hoare-based program logic using SMT solvers. Unlike Raven, none of the mentioned IVLs provide native support for reasoning about concurrency. Thus, concurrency verifiers that build on these IVLs require an additional layer of encoding and extra effort by the developers to ensure overall soundness.

Similar to Raven, Boogie is a language and verifier for imperative programs whose states are first-order structures (in some background theory). Boogie's underlying program logic does not provide direct support for compositional reasoning about (heap) resources or concurrency. However, there are several verification tools for concurrent programs that build on Boogie. Two notable examples are CIVL [25] and Chalice [31]. (The latter has also been re-implemented on top of Viper.) CIVL implements a conservative extension of Boogie for verifying concurrent programs using a notion of layered refinement. Its design differs substantially from Raven's in that CIVL uses classical first-order logic (extended with linear maps [30]) rather than separation logic, and relational structured programming rather than logically atomic specifications.

Viper [37] is an IVL for reasoning about mutable heap-allocated state. It is based on implicit dynamic frames (IDF) [45], a cousin of separation logic. Viper provides two verification backends based on symbolic execution and verification condition generation (via a translation to Boogie) [24]. Notable features include support for abstract predicates, iterated separating conjunctions [36], and magic wands [8]. Several frontends for Viper target concurrent programs, including Voila [49], a proof outline checker for the concurrent separation logic TaDa [43], VerCors [6], which targets Java, C, and OpenCL among others, as well as Prusti [2,4], a verifier for (concurrent) Rust programs.

Raven owes some debt to Viper's conceptual design. In fact, the sequential subset of Raven's core language is compatible with Viper's. However, there are also important differences. (Ghost) resources in Viper are restricted to fractional permissions whereas Raven supports user-definable resource algebras. In particular, non-cancellative RAs supported by Raven but not Viper are often useful for verifying intricate concurrent algorithms (e.g., they are used in the case study discussed in §5.2). Also, Viper has no module system. At a more technical level, Raven's SMT encoding of resource reasoning can be thought of as generalizing the approach taken in Viper's VC generation backend

to arbitrary RAs. A key technical difference is that Raven adheres to separation logic rather than IDF to maintain compatibility with Iris. This creates interesting trade-offs. In separation logic, expression evaluation is not dependent on resources, which simplifies some of the concurrency reasoning compared to IDF. On the other hand, SL relies more heavily on existential quantifiers. We alleviate the problem of reasoning about existential quantifiers in Raven with a preprocessing step that computes witness terms from resources and implicit parameters of predicates. Recent efforts aim to put Viper on a foundational footing by formalizing and mechanizing its meta-theory [9] and by enabling formal validation of its SMT translation pipeline [39].

We note that VerCors, which builds on Viper may also be seen as an IVL for concurrent programming languages [1]. However, we consider Raven to be situated further down in the tool pipeline. In fact, VerCors may benefit from Raven's improved support for user-definable ghost resources and its other inbuilt concurrency reasoning features. For example, Raven's concurrency primitives are expressive enough to encode VerCors's parallel blocks directly: Raven supports spawning a thread that executes a call to a procedure `p` (which may contain the code of a parallel block). Thread spawning consumes the resources from `p`'s precondition. In addition, a shared state invariant allows transferring resources from the spawned thread back to the current thread. The examples 'fork_join' and 'fork_join_client' in our benchmark suite demonstrate this.

Steel Core [47] is an IDF-based resource logic embedded in dependent type theory. Similar to Raven, it supports user-definable partially commutative monoids, dynamically allocated invariants, and ghost computations. However, presently it does not support iterated separating conjunctions or atomic specifications. Steel Core provides foundational guarantees via a shallow embedding into F* [46]. Proof automation is enabled via the tactic engine Steel [14]. It relies on symbolic execution with a frame inference engine based on AC matching [23] and uses an SMT solver for equality reasoning modulo theories. In contrast, Raven offloads the entire reasoning to the SMT solver. By augmenting RAs with a subtraction function ($\setminus$), frames are automatically computed using theory solvers or E-matching.

VeriFast [18] is a verifier for C and Java based on separation logic with fractional permissions. It does not support user-definable RAs but can reason about fine-grained concurrency using a form of higher-order ghost code [17]. Implicit parameters of predicates in Raven are similar to VeriFast's predicate output parameters. However, implicit parameters are strictly more general than output parameters. In particular, output parameters require predicate definitions to be *precise*. Here, precise means that in any given state, there exists a unique substate that satisfies the predicate for a fixed valuation of the input parameters and this substate uniquely determines the values of the output parameters. In contrast, Raven supports imprecise predicates with implicit parameters like the `is_lock` in Fig. 3. For example, on §4, both `is_lock(l, r, false)` as well as `is_lock(l, r, true)` hold. For this reason, the user needs to specify the value of `b` to fold the predicate. However, importantly, both of these predicate instances cannot hold simultaneously (because e.g. **`own`**`(l.next, n)` is not duplicable). This correctness condition is enforced by Raven and guarantees the uniqueness of the implicit parameter values, once one of the instances has been folded.

Raven builds on the concurrent separation logic Iris [20, 22]. Iris is a higher-order impredicative logic, which necessitates a step-indexed semantic model. This is reflected in a more complex notion of RA (so-called *cameras*) and, in the form of the *later* modality in assertions. Moreover, shared invariants and judgements in Iris are annotated with namespaces and masks to soundly deal with impredicative (nested) invariants. These features add complexity when learning the logic, and considerable proof burden even for simple examples. A Raven RA roughly corresponds to a discrete unital camera in Iris. To facilitate proof automation and accessibility, we restrict Iris to its first-order subset by disallowing higher-order quantification and impredicativity. The simplified setting avoids the need for explicit reasoning about Iris's step-indexed semantics. Also, invariants in Raven impose restrictions on the structure of namespaces so that they are not directly exposed to the user and mask annotations can be automatically inferred. To compensate for the resulting loss of expressivity, Raven provides a higher-order module system, which can be used, e.g., to quantify over abstract predicates (like `resource` in Fig. 1a). Iris provides tooling support via integration with the Rocq proof assistant [26]. Recent work has extended the Rocq plugin with a tactics-based proof search engine, called Diaframe [34, 35], which we discuss in more detail in §5.

Several earlier notable concurrent separation logics that informed the development of Iris have been used as the foundation for tool development efforts. For instance, CAP [11] has been implemented in Caper [12], the views framework [10] in Starling [48], and TaDa [43] in Voila. Proofs conducted with these tools are either restricted to using specific notions of resources or provide less automation than Raven.

7 Conclusions

We introduce Raven, an intermediate language and tool for deductive verification of concurrent programs. Raven is based on the concurrent separation logic Iris, but carefully restricts its expressivity to enable proof automation via SMT solvers. Our experimental comparison shows that Raven is significantly faster and provides a better user experience for larger proof efforts compared to other existing proof automation tools targeting Iris.

We also have a formalization of Raven's program logic for a core fragment of its programming language as well as a pencil and paper soundness proof of the verification condition generator for this fragment. A Rocq mechanization that embeds Raven's program logic into an Iris instance is underway. However, this is outside the scope of the present paper.

We plan to expand Raven's ecosystem by integrating it into front-end verification tools. Other future work includes the integration of prophecies [21] to reason about future-dependent linearization points and exploring techniques for further improving proof automation, e.g., by using ghost state morphisms [38] to automatically infer frame-preserving updates.

Acknowledgments. This work is supported in parts by the National Science Foundation under the grant agreement CCF-2304758.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Armborst, L., Bos, P., van den Haak, L.B., Huisman, M., Rubbens, R., Sakar, Ö., Tasche, P.: The vercors verifier: A progress report. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II. Lecture Notes in Computer Science, vol. 14682, pp. 3–18. Springer (2024). https://doi.org/10.1007/978-3-031-65630-9_1, https://doi.org/10.1007/978-3-031-65630-9_1
2. Astrauskas, V., Bílý, A., Fiala, J., Grannan, Z., Matheja, C., Müller, P., Poli, F., Summers, A.J.: The Prusti project: Formal verification for Rust. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13260, pp. 88–108. Springer (2022). https://doi.org/10.1007/978-3-031-06773-0_5, https://doi.org/10.1007/978-3-031-06773-0_5
3. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17, https://doi.org/10.1007/11804192_17
4. Bílý, A., Pereira, J.C., Schär, J., Müller, P.: Refinement proofs in Rust using ghost locks. CoRR **abs/2311.14452** (2023). https://doi.org/10.48550/ARXIV.2311.14452, https://doi.org/10.48550/arXiv.2311.14452
5. Bizjak, A., Gratzer, D., Krebbers, R., Birkedal, L.: Iron: Managing obligations in higher-order concurrent separation logic. Proc. ACM Program. Lang. **3**(POPL), 65:1–65:30 (2019). https://doi.org/10.1145/3290378, https://doi.org/10.1145/3290378
6. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The vercors tool set: Verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) Integrated Formal Methods. pp. 102–110. Springer International Publishing, Cham (2017)
7. Bornat, R., Calcagno, C., O'Hearn, P.W., Parkinson, M.J.: Permission accounting in separation logic. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005. pp. 259–270. ACM (2005). https://doi.org/10.1145/1040305.1040327, https://doi.org/10.1145/1040305.1040327
8. Dardinier, T., Parthasarathy, G., Weeks, N., Müller, P., Summers, A.J.: Sound automation of magic wands. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 130–151. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_7, https://doi.org/10.1007/978-3-031-13188-2_7
9. Dardinier, T., Sammler, M., Parthasarathy, G., Summers, A.J., Müller, P.: Formal foundations for translational separation logic verifiers. Proc. ACM Program. Lang. **9**(POPL) (Jan 2025). https://doi.org/10.1145/3704856, https://doi.org/10.1145/3704856
10. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M.J., Yang, H.: Views: Compositional reasoning for concurrent programs. In: Giacobazzi, R., Cousot, R. (eds.) The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 287–300. ACM (2013). https://doi.org/10.1145/2429069.2429104, https://doi.org/10.1145/2429069.2429104
11. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: D'Hondt, T. (ed.) ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings. Lecture Notes in

Computer Science, vol. 6183, pp. 504–528. Springer (2010). https://doi.org/10.1007/978-3-642-14107-2_24, https://doi.org/10.1007/978-3-642-14107-2_24
12. Dinsdale-Young, T., da Rocha Pinto, P., Andersen, K.J., Birkedal, L.: Caper - automatic verification for fine-grained concurrency. In: Yang, H. (ed.) Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10201, pp. 420–447. Springer (2017). https://doi.org/10.1007/978-3-662-54434-1_16, https://doi.org/10.1007/978-3-662-54434-1_16
13. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8, https://doi.org/10.1007/978-3-642-37036-6_8
14. Fromherz, A., Rastogi, A., Swamy, N., Gibson, S., Martínez, G., Merigoux, D., Ramananandro, T.: Steel: Proof-oriented programming in a dependently typed concurrent separation logic. Proc. ACM Program. Lang. **5**(ICFP), 1–30 (2021). https://doi.org/10.1145/3473590, https://doi.org/10.1145/3473590
15. Gäher, L., Sammler, M., Jung, R., Krebbers, R., Dreyer, D.: RefinedRust: A type system for high-assurance verification of Rust programs. Proc. ACM Program. Lang. **8**(PLDI) (Jun 2024). https://doi.org/10.1145/3656422, https://doi.org/10.1145/3656422
16. Gupta, E., Nisarg, P., Wies, T.: Raven: An SMT-based concurrency verifier (May 2025). https://doi.org/10.5281/zenodo.15477369, https://doi.org/10.5281/zenodo.15477369
17. Jacobs, B., Piessens, F.: Expressive modular fine-grained concurrency specification. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 271–282. ACM (2011). https://doi.org/10.1145/1926385.1926417, https://doi.org/10.1145/1926385.1926417
18. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4, https://doi.org/10.1007/978-3-642-20398-5_4
19. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: Securing the foundations of the Rust programming language. Proceedings of the ACM on Programming Languages **2**(POPL), 1–34 (2017)
20. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018). https://doi.org/10.1017/S0956796818000151, https://doi.org/10.1017/S0956796818000151
21. Jung, R., Lepigre, R., Parthasarathy, G., Rapoport, M., Timany, A., Dreyer, D., Jacobs, B.: The future is ours: Prophecy variables in separation logic. Proc. ACM Program. Lang. **4**(POPL), 45:1–45:32 (2020). https://doi.org/10.1145/3371113, https://doi.org/10.1145/3371113
22. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January

15-17, 2015. pp. 637–650. ACM (2015). https://doi.org/10.1145/2676726.2676980, https://doi.org/10.1145/2676726.2676980
23. Kapur, D., Narendran, P.: Complexity of unification problems with associative-commutative operators. J. Autom. Reason. **9**(2), 261–288 (1992). https://doi.org/10.1007/BF00245463, https://doi.org/10.1007/BF00245463
24. Kassios, I.T., Müller, P., Schwerhoff, M.: Comparing verification condition generation with symbolic execution: An experience report. In: Joshi, R., Müller, P., Podelski, A. (eds.) Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7152, pp. 196–208. Springer (2012). https://doi.org/10.1007/978-3-642-27705-4_16, https://doi.org/10.1007/978-3-642-27705-4_16
25. Kragl, B., Qadeer, S.: The Civl verifier. In: Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021. pp. 143–152. IEEE (2021). https://doi.org/10.34727/2021/ISBN.978-3-85448-046-4_23, https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_23
26. Krebbers, R., Jourdan, J., Jung, R., Tassarotti, J., Kaiser, J., Timany, A., Charguéraud, A., Dreyer, D.: Mosel: A general, extensible modal framework for interactive proofs in separation logic. Proc. ACM Program. Lang. **2**(ICFP), 77:1–77:30 (2018). https://doi.org/10.1145/3236772, https://doi.org/10.1145/3236772
27. Krishna, S., Patel, N., Shasha, D.E., Wies, T.: Verifying concurrent search structure templates. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 181–196. ACM (2020). https://doi.org/10.1145/3385412.3386029, https://doi.org/10.1145/3385412.3386029
28. Krishna, S., Shasha, D.E., Wies, T.: Go with the flow: Compositional abstractions for concurrent data structures. Proc. ACM Program. Lang. **2**(POPL), 37:1–37:31 (2018). https://doi.org/10.1145/3158125, https://doi.org/10.1145/3158125
29. Krishna, S., Summers, A.J., Wies, T.: Local reasoning for global graph properties. In: Müller, P. (ed.) Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12075, pp. 308–335. Springer (2020). https://doi.org/10.1007/978-3-030-44914-8_12, https://doi.org/10.1007/978-3-030-44914-8_12
30. Lahiri, S.K., Qadeer, S., Walker, D.: Linear maps. In: Jhala, R., Swierstra, W. (eds.) Proceedings of the 5th ACM Workshop Programming Languages meets Program Verification, PLPV 2011, Austin, TX, USA, January 29, 2011. pp. 3–14. ACM (2011). https://doi.org/10.1145/1929529.1929531, https://doi.org/10.1145/1929529.1929531
31. Leino, K.R.M., Müller, P., Smans, J.: Verification of concurrent programs with Chalice. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures. Lecture Notes in Computer Science, vol. 5705, pp. 195–222. Springer (2009). https://doi.org/10.1007/978-3-642-03829-7_7, https://doi.org/10.1007/978-3-642-03829-7_7
32. Mansky, W., Du, K.: An Iris instance for verifying CompCert C programs **8**(POPL) (Jan 2024). https://doi.org/10.1145/3632848, https://doi.org/10.1145/3632848
33. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24, https://doi.org/10.1007/978-3-540-78800-3_24

34. Mulder, I., Krebbers, R.: Proof automation for linearizability in separation logic. Proc. ACM Program. Lang. **7**(OOPSLA1), 462–491 (2023). https://doi.org/10.1145/3586043, https://doi.org/10.1145/3586043
35. Mulder, I., Krebbers, R., Geuvers, H.: Diaframe: Automated verification of fine-grained concurrent programs in Iris. In: Jhala, R., Dillig, I. (eds.) PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022. pp. 809–824. ACM (2022). https://doi.org/10.1145/3519939.3523432, https://doi.org/10.1145/3519939.3523432
36. Müller, P., Schwerhoff, M., Summers, A.J.: Automatic verification of iterated separating conjunctions using symbolic execution. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9779, pp. 405–425. Springer (2016). https://doi.org/10.1007/978-3-319-41528-4_22, https://doi.org/10.1007/978-3-319-41528-4_22
37. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings. Lecture Notes in Computer Science, vol. 9583, pp. 41–62. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_2, https://doi.org/10.1007/978-3-662-49122-5_2
38. Nanevski, A., Banerjee, A., Delbianco, G.A., Fábregas, I.: Specifying concurrent programs in separation logic: Morphisms and simulations. Proc. ACM Program. Lang. **3**(OOPSLA), 161:1–161:30 (2019). https://doi.org/10.1145/3360587, https://doi.org/10.1145/3360587
39. Parthasarathy, G., Müller, P., Summers, A.J.: Formally validating a practical verification condition generator. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 704–727. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_33, https://doi.org/10.1007/978-3-030-81688-9_33
40. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 773–789. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_54, https://doi.org/10.1007/978-3-642-39799-8_54
41. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic with trees and data. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 711–728. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_47, https://doi.org/10.1007/978-3-319-08867-9_47
42. Piskac, R., Wies, T., Zufferey, D.: Grasshopper - complete heap verification with mixed specifications. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8413, pp. 124–139. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_9, https://doi.org/10.1007/978-3-642-54862-8_9
43. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: Tada: A logic for time and data abstraction. In: Jones, R.E. (ed.) ECOOP 2014 - Object-Oriented Programming - 28th European

Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8586, pp. 207–231. Springer (2014). https://doi.org/10.1007/978-3-662-44202-9_9, https://doi.org/10.1007/978-3-662-44202-9_9

44. Schwerhoff, M.: Voila GitHub issue #33. https://github.com/viperproject/voila/issues/33, last accessed: April 15, 2025
45. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. ACM Trans. Program. Lang. Syst. **34**(1), 2:1–2:58 (2012). https://doi.org/10.1145/2160910.2160911, https://doi.org/10.1145/2160910.2160911
46. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 256–270. ACM (2016). https://doi.org/10.1145/2837614.2837655, https://doi.org/10.1145/2837614.2837655
47. Swamy, N., Rastogi, A., Fromherz, A., Merigoux, D., Ahman, D., Martínez, G.: SteelCore: an extensible concurrent separation logic for effectful dependently typed programs. Proc. ACM Program. Lang. **4**(ICFP), 121:1–121:30 (2020). https://doi.org/10.1145/3409003, https://doi.org/10.1145/3409003
48. Windsor, M., Dodds, M., Simner, B., Parkinson, M.J.: Starling: Lightweight concurrency verification with views. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10426, pp. 544–569. Springer (2017). https://doi.org/10.1007/978-3-319-63387-9_27, https://doi.org/10.1007/978-3-319-63387-9_27
49. Wolf, F.A., Schwerhoff, M., Müller, P.: Concise outlines for a complex logic: A proof outline checker for tada. Formal Methods Syst. Des. **61**(1), 110–136 (2022). https://doi.org/10.1007/S10703-023-00427-W, https://doi.org/10.1007/s10703-023-00427-w