Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA

Felix A. Wolf, Malte Schwerhoff, and Peter Müller

Department of Computer Science, ETH Zurich
{felix.wolf,malte.schwerhoff,peter.mueller}@inf.ethz.ch

Abstract. Modern separation logics allow one to prove rich properties of intricate code, e.g. functional correctness and linearizability of non-blocking concurrent code. However, this expressiveness leads to a complexity that makes these logics difficult to apply. Manual proofs or proofs in interactive theorem provers consist of a large number of steps, often with subtle side conditions. On the other hand, automation with dedicated verifiers typically requires sophisticated proof search algorithms that are specific to the given program logic, resulting in limited tool support that makes it difficult to experiment with program logics, e.g. when learning, improving, or comparing them. Proof outline checkers fill this gap. Their input is a program annotated with the most essential proof steps, just like the proof outlines typically presented in papers. The tool then checks automatically that this outline represents a valid proof in the program logic. In this paper, we systematically develop a proof outline checker for the TaDA logic, which reduces the checking to a simpler verification problem, for which automated tools exist. Our approach leads to proof outline checkers that provide substantially more automation than interactive provers, but are much simpler to develop than custom automatic verifiers.

1 Introduction

Standard separation logic enables the modular verification of heap-manipulating sequential [25,33] and data-race free concurrent programs [24,4]. More recently, numerous separation logics have been proposed that enable the verification of fine-grained concurrency by incorporating ideas from concurrent separation logic, Owicki-Gries [28], and rely-guarantee [15]. Examples include CAP [7], iCAP [40], CaReSL [42], CoLoSL [32], FCSL [37], GPS [43], RSL [45], and TaDA [35] (see Brookes et al. [3] for an overview). These logics are very expressive, but challenging to apply because they often comprise many complex proof rules. E.g. our running example (Fig. 1) consists of two statements, but requires over 20 rule applications in TaDA, many of which have non-trivial instantiations and subtle side conditions. This complexity seems inevitable for challenging verification problems involving, e.g. fine-grained concurrency or weak memory.

The complexity of advanced separation logics makes it difficult to develop proofs in these logics. It is, thus, crucial to have tools that check the validity

of proofs and automate parts of the proof search. One way to provide this tool support is through *proof checkers*, which take as input a nearly complete proof and check its validity. They typically embed program logics into the higher-order logic of an interactive theorem prover such as Coq. Proof checkers exist, e.g. for RSL [45] and FCSL [37]. Alternatively, *automated verifiers* take as input a program with specifications and devise the proof automatically. They typically combine existing reasoning engines such as SMT solvers with logic-specific proof search algorithms. Examples are Smallfoot [2] and Grasshopper [31] for traditional separation logics, and Caper [8] for fine-grained concurrency.

Proof checkers and automated verifiers strike different trade-offs in the design space. Proof checkers are typically very expressive, enabling the verification of complex programs and properties, and produce foundational proofs. However, existing proof checkers offer little automation. Automated verifiers, on the other hand, significantly reduce the proof effort, but compromise on expressiveness and require substantial development effort, especially, to devise custom proof search algorithms.

It is in principle possible to increase the automation of proof checkers by developing proof tactics, or to increase the expressiveness of automated verifiers by developing stronger custom proof search algorithms. However, such developments are too costly for the vast majority of program logics, which serve mostly a scientific or educational purpose. As a result, adequate tool support is very rare, which makes it difficult for developers of such logics, lecturers and students, as well as engineers to apply, and gain experience with, such logics.

To remedy the situation, several tools took inspiration from the idea of *proof outlines* [27,1], formal proof skeletons that contain the key proof steps, but omit most of the details. Proof outlines are a standard notation to present program proofs in publications and teaching material. *Proof outline checkers* such as Starling [46] and VeriFast [14] take as input a proof outline and then check automatically that it represents a valid proof in the program logic. They provide automation for proof steps for which good proof search algorithms exist, and can support expressive logics by requiring annotations for complex proof steps. Due to this flexibility, proof outline checkers are especially useful for experimenting with a logic, in situations where foundational proofs are not essential.

In this paper, we present Voila, a proof outline checker for TaDA [35], which goes beyond existing proof outline checkers and automated verifiers by supporting a substantially more complex program logic, handling fine-grained concurrency, linearizability, abstract atomicity, and other advanced features. We believe that our systematic development of Voila generalizes to other complex logics. Our contributions are as follows:

- The Voila *proof outline language*, which supports a large subset of TaDA and enables users to write proof outlines very similar to those used by the TaDA authors [35,34] (Sec. 3).
- A systematic approach to automate the expansion of a proof outline into a full *proof candidate* via a normal form and heuristics (Sec. 5). Our approach automates most proof steps (20 out of 22 in the running example from Fig. 1).

- An encoding of the proof candidate into Viper [22], which checks its validity without requiring any TaDA-specific proof search algorithms (Sec. 6).
- The Voila proof outline checker, the *first* tool that supports specification for linearization points, provides a high degree of automation, and achieves good performance (Sec. 7). Our submission artifact with the Voila tool ready-to-use can be found at [48], and the Voila source repository is located at [47].

Outline. Sec. 2 gives an overview of the TaDA logic and illustrates our approach. Sec. 3 presents the Voila proof outline language, and Sec. 4 summarizes how we verify proof outlines. We explain how we automatically expand a proof outline into a proof candidate in Sec. 5 and how we encode a proof candidate into Viper in Sec. 6. In Sec. 7, we evaluate our technique by verifying several challenging examples, discuss related work in Sec. 8, and conclude in Sec. 9.

The full version of our paper [49] contains a substantial appendix with many further details, including: the full version and Viper encoding of our running example, with TaDA levels (omitted from this paper, but supported by Voila) and nested regions; additional inference heuristics; general Viper encoding scheme; encoding of a custom guard algebra; and a substantial soundness sketch.

2 Running Example and TaDA Overview

Fig. 1 shows our running example: a TaDA proof outline for the `lock` procedure of a spinlock. As in the original publication [35], the outline shows only two out of 22 proof steps and omits most side conditions. We use this example to introduce the necessary TaDA background, explain TaDA proof outlines, and illustrate the corresponding Voila proof outline.

2.1 Regions and Atomicity

TaDA targets shared-memory concurrency with sequentially consistent memory. TaDA programs manipulate *shared regions*, data structures that are concurrently modified according to a specified *protocol* (as in rely-guarantee reasoning [15]). A shared region such as $\texttt{Lock}_r(\texttt{x}, s)$ is an abstraction over the region's content, analogous to abstract predicates [30] in traditional separation logic. In our example (lines 1–2), the lock owns memory location x (denoted by separation logic's points-to predicate $x \mapsto \_$), and its *abstract state* s is 0 or 1, indicating whether it is unlocked or locked. Here, the abstract state and the content of the memory location coincide, but they may differ in general. The subscript r uniquely identifies a region instance. Note that TaDA's region assertions are duplicable, such that multiple threads may obtain an instance of the $\texttt{Lock}_r$ resource and invoke operations on the lock.

Lines 3–5 define the protocol for modifications of a lock as a labeled transition system. The labels are *guards* – abstract resources that restrict when a transition may be taken. Here, guard G allows both locking and unlocking (lines 3-4), and is unique (line 5). Most lock specifications use duplicable guards to allow multiple

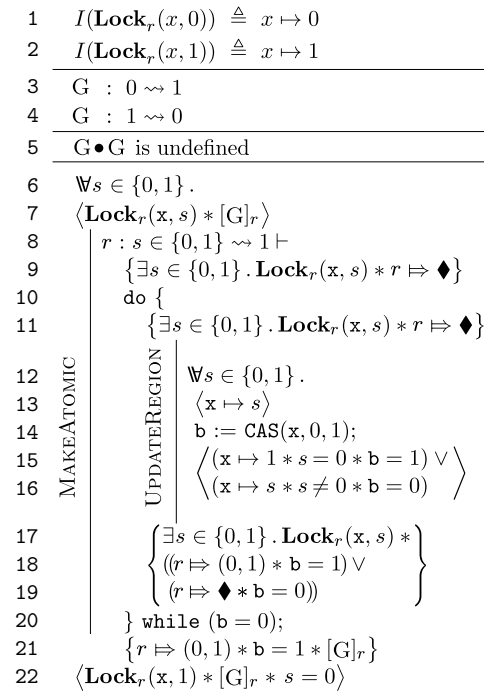

Fig. 1: TaDA spinlock example with shared region `Lock`; adapted with only minor changes from TaDA [35]. The lock region (lines 1–2) comprises a single memory location, whose value is either 0 (available) or 1 (acquired). Guard G allows locking and unlocking (lines 3–4), and is unique (line 5). The proof outline (lines 6–22) shows a CAS-based `lock` operation with atomic specifications. An enclosing region (`CAPLock` in da Rocha Pinto et al. [35], verifiable by Voila and shown in the full paper [49] then establishes the usual lock semantics. Levels (denoted by λ in TaDA) are omitted from the discussion in this paper, but supported by Voila and included in the full paper [49].

threads to compete for the lock; in this example, the usual lock semantics is established by an enclosing region (`CAPLock` [35]; see the full paper [49]).

Lines 6–22 contain the proof outline for the `lock` procedure, which updates a lock `x` from an undetermined state – it can seesaw between locked and unlocked due to environment interference – to the locked state. Importantly, this update appears to be atomic to clients of the spinlock. These properties are expressed by the *atomic TaDA triple* (lines 6, 7, and 22)

$$\forall s \in \{0,1\} \cdot \langle \mathtt{Lock}_r(\mathtt{x}, s) * [\mathrm{G}]_r \rangle \ \mathtt{lock(x)} \ \langle \mathtt{Lock}_r(\mathtt{x}, 1) * [\mathrm{G}]_r * s = 0 \rangle$$

Atomic triples (angle brackets) express that their statement is linearizable [13]. The abstract state of shared regions occurring in pre- and postconditions of atomic triples is interpreted relative to the linearization point, i.e. the moment in time when the update becomes visible to other threads (here, when the CAS operation on line 14 succeeds). The *interference context* $\forall s \in \{0,1\}$ is a special binding for the abstract region state that forces callers to guarantee that the environment keeps the lock state in $\{0,1\}$ until the linearization point is reached (a vacuous restriction in this case).

The precondition of the triple states that an instance of guard G for region r, $[\mathrm{G}]_r$, is required to execute `lock(x)`. The postcondition expresses that, at the linearization point, the lock's abstract state was changed from unlocked ($s = 0$) to locked ($\mathtt{Lock}_r(\mathtt{x}, 1)$). In general, callers must assume that a region's abstract state may have been changed by the environment after the linearization point

$$\textsc{MakeAtomic}\quad \frac{r \notin \mathcal{A} \qquad \{(x,y) \mid x \in X, y \in Y\} \subseteq \mathcal{T}_{\mathrm{R}}(\mathrm{G})^* \qquad r : x \in X \rightsquigarrow Y, \mathcal{A} \vdash \left\{\exists x \in X.\, \mathbf{R}^{\boldsymbol{\lambda}}_r(\vec{z}, x) * r \Mapsto \blacklozenge\right\} \mathbb{C} \left\{\exists x \in X, y \in Y.\, r \Mapsto (x,y)\right\}}{\mathcal{A} \vdash \mathbb{\forall} x \in X. \left\langle \mathbf{R}^{\boldsymbol{\lambda}}_r(\vec{z}, x) * [\mathrm{G}]_r \right\rangle \mathbb{C} \left\langle \ \exists y \in Y.\, \mathbf{R}^{\boldsymbol{\lambda}}_r(\vec{z}, y) * [\mathrm{G}]_r \right\rangle}$$

$$\textsc{UpdateRegion}\quad \frac{\mathcal{A} \vdash \mathbb{\forall} x \in X. \left\langle I(\mathbf{R}^{\boldsymbol{\lambda}}_r(\vec{z}, x)) * P(x) \right\rangle \mathbb{C} \left\langle \exists y \in Y, w \in W. \begin{array}{l} I(\mathbf{R}^{\boldsymbol{\lambda}}_r(\vec{z}, y)) * Q_1(x,y,w) \\ \vee\ I(\mathbf{R}^{\boldsymbol{\lambda}}_r(\vec{z}\ x)) * Q_2(x,w) \end{array} \right\rangle}{\begin{array}{l} r : x \in X \rightsquigarrow Y, \mathcal{A} \vdash \mathbb{\forall} x \in X. \left\langle \mathbf{R}^{\boldsymbol{\lambda}}_r(\vec{z}, x) * P(x) * r \Mapsto \blacklozenge \right\rangle \\ \qquad\qquad \mathbb{C} \\ \qquad \left\langle \exists y \in Y, w \in W. \begin{array}{l} \mathbf{R}^{\boldsymbol{\lambda}}_r(\vec{z}, y) * r \Mapsto (x,y) * Q_1(x,y,w) \\ \vee\ \mathbf{R}^{\boldsymbol{\lambda}}_r(\vec{z}, x) * r \Mapsto \blacklozenge \qquad * Q_2(x,w) \end{array} \right\rangle \end{array}}$$

Fig. 2: Simplified versions of two key TaDA rules used in Fig. 1. MakeAtomic establishes an atomic triple (conclusion) for a linearizable block of code (premise), which includes checking that a state update complies with the region's transition system: $\mathcal{T}_R(G)^*$ is the reflexive, transitive closure of the transitions that G allows. UpdateRegion identifies a linearization point, for instance, a CAS statement. If successful, the diamond tracking resource $r \Mapsto \blacklozenge$ is exchanged for the witness tracking resource $r \Mapsto (x, y)$ to record the performed state update; otherwise, the diamond resource is kept, such that the operation can be attempted again.

was reached; here, however, the presence of the unique guard $[\mathrm{G}]_r$ enables the caller of `lock` to conclude (by the transition system) that the lock remains locked.

2.2 TaDA Proof Outline

Lines 6–22 of the proof outline in Fig. 1 show the main proof steps; Fig. 2 shows simplified versions of the applied key TaDA rules. MakeAtomic establishes an atomic triple by checking that a block of code is atomic w.r.t. a shared region abstraction (hence the change from non-atomic premise triple, written with curly braces, to an atomic conclusion triple). UpdateRegion identifies the linearization point inside this code block. Rule MakeAtomic requires that the *atomicity context*, a set $\mathcal{A}$ of *pending updates*, of the premise triple includes any region updates performed by the statement of the triple (there can be at most one such update per region). In the proof outline, this requirement is reflected on line 8, which shows the intended update of the lock's state: $r : s \in \{0, 1\} \rightsquigarrow 1$ (following TaDA publications, we omitted the tail of the atomicity context from the outline). MakeAtomic checks that the update is allowed by the region's transition system with the available guards (the rule's second premise in Fig. 2), but the check is omitted from the proof outline. Then MakeAtomic temporarily exchanges the corresponding guard $[\mathrm{G}]_r$ for the *diamond tracking resource* $r \Mapsto \blacklozenge$ (line 9), which serves as evidence that the intended update was not yet performed.

Inside the loop, an application of UpdateRegion identifies the CAS (line 14) as the linearization point. The rule requires the diamond resource in its precondition (line 11), modifies the shared region (lines 12–16), and case-splits in its

postcondition: if the update failed (line 19) then the diamond is kept for the next attempt; otherwise (line 18), the diamond is exchanged for the *witness tracking resource* $r \Rightarrow (0, 1)$, which indicates that the region was updated from abstract state 0 to 1. At the end of MakeAtomic (lines 21–22), the witness resource is consumed and the desired abstractly atomic postcondition is established, stating that the shared region was updated from 0 to 1 at the linearization point.

2.3 Voila Proof Outline

Fig. 3 shows the *complete* proof outline of our example in the Voila proof outline language, which closely resembles the TaDA outline from Fig. 1. In particular, the `region` declaration defines a region's interpretation, abstract state, and transition system, just like the initial declarations in Fig. 1. The subsequent proof outline for procedure `lock` annotates the same two rule applications as the TaDA outline and a very similar loop invariant. The Voila proof outline verifies automatically via an encoding into Viper, but the outline is expressed completely in terms of TaDA concepts; it does not expose any details of the underlying verification infrastructure. This means that our tool automatically infers the additional 20 rule applications, and all omitted side conditions, thereby closing the gap between the user-provided proof outline and a corresponding full-fledged proof.

3 Proof Outline Language

Proof outlines annotate programs with rule applications of a given program logic. These annotations indicate where to apply rules and how to instantiate their meta-variables. The goal of a proof outline is to convey the essential proof steps; ideally, consumers of such outlines can then construct a full proof with modest effort. Consumers may be human readers [27], or tools that automatically check the validity of a proof outline [14,21,46]; our focus is on the latter.

The key challenge of designing a proof outline language is to define annotations that accomplish this goal with low annotation overhead for proof outline authors. To approach this challenge systematically, we classify the rules of the program logic (here: TaDA) into three categories: (1) For some rules, the program prescribes where and how to apply them, i.e. they do not require any annotations. We call such rules *syntax-driven rules*. An example in standard Hoare logic is the assignment rule, where the assignment statement prescribes how to manipulate the adjacent assertions. (2) Some rules can be applied and instantiated in many meaningful ways. For such rules, the author of the proof outline needs to indicate where or how to apply them through suitable annotations. Since such rules often indicate essential proof steps, we call them *key rules*. In proof outlines for standard Hoare logic, the while-rule typically requires an annotation *how* to apply it, namely the loop invariant. The rule of consequence typically requires an annotation *where and how* to apply it, e.g. to strengthen the precondition of a triple or to weaken its postcondition. (3) The effort of authoring a proof outline can be greatly reduced by applying some rules heuristically, based on information

```
struct cell { int val; }

region Lock(id r, cell x)
  interpretation { x.val |-> ?v && (v == 0 || v == 1) }
  state { v }
  guards { unique G; }
  actions { G: 0 ~> 1; G: 1 ~> 0; }

abstract_atomic procedure lock(id r, cell x)
  interference ?s in Set(0, 1);
  requires Lock(r, x, s) && G@r;
  ensures  Lock(r, x, 1) && G@r && s == 0;
{
  bool b;
  make_atomic using Lock(r, x) with G@r {
    do
      invariant Lock(r, x);
      invariant !b ==> r |=> <D>;
      invariant  b ==> r |=> (0, 1);
    {
      update_region using Lock(r, x) {
        b := CAS(x, 0, 1);
      }
    } while (!b);
  }
}
```

Fig. 3: The Voila proof outline of our example, strongly resembling the TaDA proof outline from Fig. 1. `id` is the type of region identifiers; primitive types are passed by value, structs by reference. Logical variables are introduced using a question mark; e.g. `x.val`$\mapsto$`?v` binds the logical variable `v` to the value of the location `x.val`. `&&` denotes separating conjunction.

already present in the outline. We call such rules *bridge rules*. Heuristics reduce the annotation overhead, but may lead to incompleteness if they fail; a proof outline language may provide annotations to complement the heuristics in such situations, slightly blurring the distinction between key and bridge rules. E.g. the Dafny verifier [20] applies heuristics to guess termination measures for loops, but also offers an annotation to provide a measure manually, if necessary.

The rule classification depends on the proof search capabilities of the verification tool that is used to check the proof outline. We use Viper [22], which provides a high degree of automation for standard separation logic and, thus, allows us to focus on the specific aspects of TaDA.

In the rest of this section, we give an overview of the Voila proof outline language and, in particular, discuss which TaDA rules are supported as syntax-driven, key, and bridge rules. Voila's grammar can be found in the full paper [49], showing that Voila strongly resembles TaDA, but requires fewer technical details.

Expressions and Statements. Voila supports all of TaDA's programming language constructs, including variables and heap locations, primitive types and operations thereon, atomic heap reads and writes, loops, and procedure calls. Consequently, Voila supports the corresponding syntax-driven TaDA rules.

Background Definitions. Voila's syntax for declaring regions and transitions closely resembles TaDA, but e.g. subscripts are replaced by additional parameters, such as the region identifier `r`. A region declaration defines the region's content via an `interpretation` assertion, and its value via a `state` function. The latter may refer to region parameters, as well as values bound in the interpretation, such as `v` in the example from Fig. 3. The region's transition system is declared by introducing the guards and the permitted *actions*, i.e. transitions. Voila includes several built-in guard algebras (adopted from Caper [8]); additional ones can be encoded, see the full paper [49]. A region declaration introduces a corresponding region predicate, which has an additional out-parameter that yields the region's abstract state (e.g. `s` in the precondition of procedure `lock` in Fig. 3), as defined by the `state` function. We omit this out-parameter when its value is irrelevant.

Specifications. Voila proof outlines require specifications for procedures, and invariants for loops; we again chose a TaDA-like syntax for familiarity. Explicit loop invariants are required by Viper, but also enable us to automatically instantiate certain bridge rules (see framing in Sec. 5).

Recall that specifications in TaDA are written as atomic or non-atomic triples, and include an interference context and an atomicity context. Voila simplifies the notation significantly by requiring these contexts only for abstractly-atomic procedure specifications; for all statements and rule applications, they are determined automatically, despite changing regularly during a proof. For procedures with abstractly-atomic behavior (modifier `abstract_atomic`), the interference context is declared through the `interference` clause. E.g. for procedure `lock` from Fig. 3, it corresponds to TaDA's interference context $\mathbb{W}s \in \{0, 1\}$.

Key Rules. In addition to procedure and loop specifications, Voila requires user input only for the following fundamental TaDA rules: UpdateRegion, MakeAtomic, UseAtomic, and OpenRegion; applications of all other rules are automated. Since they capture the core ideas behind TaDA, these rules are among the most complex rules of the logic and admit a vast proof search space. Therefore, their annotation is essential, for both human readers [35,34] and automatic checkers. As seen in Fig. 3, the annotations for these key rules include only the used region and, for updates, the used guard; all other information present in the corresponding TaDA rules is derived automatically.

Bridge Rules. All other TaDA rules are applied automatically, and thus have no Voila counterparts. This includes all structural rules for manipulating triple atomicity (e.g. AWeakening1, AExists), interference contexts (e.g. Substitution, AWeakening2), and levels (e.g. AWeakening3). Their applications are heuristically derived from the program, applications of key rules, and adjacent triples. TaDA's frame rule is also automatically applied by leveraging Viper's built-in support for framing, combined with additional encoding steps to satisfy TaDA's frame stability side condition. Finally, TaDA entailments are bridge rules when they can be automated by the used verification tool. For Viper, this is the case for standard separation logic entailments, which constitute the majority

of entailments to perform. To support TaDA's *view shifts* [6,34] – entailments similar to the classical rule of consequence, but involving arbitrary definitions of regions and guard algebras – Voila provides specialized annotations.

4 Proof Workflow

Our approach, and corresponding implementation, enables the following workflow: users provide a proof outline and possibly some annotations for complex entailments, but never need to insert any other rule. Hence, if the outline summarizes a valid proof, verification is automatic, without a tedious process of manually applying additional rules. If the outline is invalid, our tool reports which specification (e.g. loop invariant) it could not prove or which key rule application it could not verify, and why (e.g. missing guard).

Achieving this workflow, however, is challenging: by design, proof outlines provide the important proof steps, but are not complete proofs. Consider, e.g. the TaDA and Voila outlines from Fig. 1 and Fig. 3, respectively. Applying UpdateRegion produces an atomic triple in its conclusion, whereas the while-rule requires a non-atomic triple for the loop body. A complete proof needs to perform the necessary adjustment through additional applications of bridge rules, which are not present in the proof outlines, and thus need to be inferred.

Our workflow is enabled by first expanding proof outlines into *proof candidates*, in two main steps: step 1 automatically inserts the applications of all syntax-driven rules; step 2 expands further by applying heuristics to insert bridge rule applications. The resulting proof candidate contains the applications of all rules of the program logic. Afterwards, we check that the proof candidate corresponds to a valid proof, by encoding it as a Viper program that checks whether all proof rules are applied correctly. Our actual implementation deviates slightly from this conceptual structure, e.g. because Viper does not require one to make the application of syntax-driven rules, framing, and entailment checking explicit.

5 Expanding Proof Outlines to Proof Candidates

Automatically expanding a proof outline is ultimately a proof search problem, with a vast search space in case of complex logics such as TaDA. Our choice of key rules (and corresponding annotations) reduces the search space, but it remains vast, due to TaDA's many structural rules that can be applied to almost all triples. To further reduce the search space, without introducing additional annotation overhead, we devised (and enforce) a *normal form* for proof candidate triples. Our normal form allows us to define *heuristics* for the application of bridge rules *locally*, based only on adjacent rule applications, without having to inspect larger proof parts. This locality reduces the search space substantially, and enables us to automatically close the gap between user-provided proof outline and finally verified proof candidate. In our running example, our heuristics infer 20 out of 22 rule applications.

It might be helpful to consider an analogy with standard Hoare logic: its rule of consequence can be applied to each Hoare triple. A suitable normal form could restrict proofs to use the rule of consequence only at the beginning of the program and for each loop (as in a weakest-precondition calculus). A heuristic can then infer the concrete applications, in particular, the entailments used in the rule application, treating the rule as a bridge rule.

Normal Form. Our normal is established by a combination of syntactic checks and proof obligations in the final Viper encoding. Its main restrictions are as follows: (1) All triples are either exclusively atomic or non-atomic, which enables us to infer the triple kinds from statements and key rule applications. Due to this restriction, Voila cannot express specifications that combine atomic and non-atomic behaviors. However, such specifications do not occur frequently (see Sec. 5.2.3 in [34] for an example) and could be supported via additional annotations. (2) All triple preconditions, as well as the postconditions of non-atomic triples, are *stable*, i.e. cannot be invalidated by (legal) concurrent operations. In contrast, TaDA requires stability only for certain assertions. Our stronger requirement enables us to *rely* on stability at various points in the proof instead of having to *check* it – most importantly, when Viper automatically applies its frame rule. To enforce this restriction, we eagerly stabilize assertions through suitable weakening steps. (3) In atomic triples, the state of every region is bound by exactly one interference quantifier ($\mathbb{V}$), which simplifies the manipulation of interference contexts, e.g. for procedure calls. To the best of our knowledge, this restriction does not limit the expressiveness of Voila proofs. (4) Triples must hold for a *range* of atomicity contexts $\mathcal{A}$, rather than just a single context. This stronger proof obligation rules out certain applications of MakeAtomic – which we have seen only in contrived examples – but it increases automation substantially and improves procedure modularity.

By design, our normal form prevents Voila from constructing certain TaDA proofs. However, the only practical limitation is that Voila does not support TaDA's combination of atomic and non-atomic behavior in a single triple. As far as we are aware, all other normal form restrictions do not limit expressiveness for practical examples, or can be worked around in systematic ways.

Heuristics. We employ five main heuristics: to determine when to change triple atomicity, to ensure stable frames by construction, to compute atomicity context ranges, to compute levels, and to compute interference contexts in procedure body proofs. All heuristics are based on inspecting adjacent rule applications and their proof state. We briefly discuss the first three heuristics here, and refer readers to the full paper [49] for the remaining two heuristics. There, we give a more detailed explanation, and illustrate our heuristics in the context of our running example. (1) Changing triple atomicity corresponds to an application of (at least) TaDA rule AWeakening1, necessary when a non-atomic composite statement (e.g. the `while` statement in Fig. 1) has an abstract-atomic sub-statement (e.g. the atomic CAS in Fig. 1). We infer all applications of this rule. (2) A more complex heuristic is used in the context of framing: TaDA's frame rule requires

the *frame*, i.e., the assertion preserved across a statement, to be stable. For simple statements such as heap accesses, it is sound to rely on Viper's built-in support for framing. For composite statements with arbitrary user-provided *footprints* (assertions such as a loop invariant describing which resources the composite statement may modify), we greedily infer frame rule applications that attempt to preserve all information outside the footprint. The inferred applications are later encoded in Viper such that the resulting frame is stable, by applying suitable weakening steps. (3) Atomicity context ranges are heuristically inferred from currently owned tracking resources and level information. Atomicity contexts are not manipulated by a specific TaDA rule, but they need to be instantiated when applying rules: most importantly, TaDA's procedure call rule, but also e.g. MakeAtomic and UpdateRegion (see Fig. 2).

In our experience, our heuristics fail *only* in two scenarios: the first are contrived examples, concerned with TaDA resources in isolation, not properties of actual code – where they fail to expand a proof outline into a valid proof. More relevant is the second scenario, where our heuristics yield a valid proof that Viper then fails to verify because it requires entailments that Viper cannot discharge automatically. To work around such problems when they occur, Voila allows programmers to provide additional annotations to indicate where to apply complex entailments.

Importantly, a failure of our heuristics does not compromise soundness: if they infer invalid bridge rule applications, e.g. whose side conditions do not hold, the resulting invalid proof candidates are rejected by Viper in the final validation.

6 Validating Proof Candidates in Viper

Proof candidates – i.e. the user-provided program with heuristically inserted bridge rule applications – do not necessarily represent valid proofs, e.g. when users provide incorrect loop invariants. To check whether a proof candidate actually represents a valid proof, we need to verify (1) that each rule is applied correctly, in particular, that its premises and side conditions hold, and (2) that the property shown by the proof candidate entails the intended specification. To validate proof candidates automatically, we use the existing Viper tool [22]. In this section, we give a high-level overview of how we encode proof candidates into the Viper language.

Viper Language. Viper uses a variation of separation logic [38,29] whose assertions separate access permissions from value information: separation logic's points-to assertion `x.f` $\mapsto$ `v` is expressed as `acc(x.f) && x.f == v`, and separation logic predicates [30] are similarly split into a predicate (abstracting over permissions) and a heap-dependent function (abstracting over values). Well-definedness checks ensure that the heap is accessed only under sufficient permissions. Viper provides a simple imperative language, which includes in particular two statements to manipulate the verification state: `exhale` A asserts all logical constraints in assertion A, removes the permissions in A from the current state (or fails if

```
⟦region R(r: id, p: t)
  interpretation I
  state S
  guards G
  actions A⟧ ≜
predicate R(r: Ref, p:⟦t⟧) { ⟦I⟧ }

function R_State(r: Ref, p:⟦t⟧): T
  requires  R(r,p)
{ unfolding R(r,p) in ⟦S⟧ }

foreach g(p': t') ∈ G:
  predicate R_g(r: Ref, p':⟦t'⟧)
end

field diamond: Bool
```

```
field val: Int

predicate Lock(r: Ref, x: Ref) {
  acc(x.val) &&
  (x.val == 0 || x.val == 1)
}

function Lock_State
            (r: Ref, x: Ref): Int
  requires Lock(r, x)
{ unfolding lock(r, x) in x.val }

predicate Lock_G(r: Ref)

field diamond: Bool
```

Fig. 4: Excerpt of the Viper encoding of regions; general case (left), and for the lock region from Fig. 3 (right). The encoding function is denoted by double square brackets; overlines denote lists; *foreach* loops are expanded statically. Type T is the type of the state expression S, which is inferred. Actions A do not induce any global declarations. The elements of struct types and type `id` are encoded as Viper references (type `Ref`). The `unfolding` expression temporarily unfolds a predicate into its definition; it is required by Viper's backend verifiers. The struct type `cell` from Fig. 3 is encoded as a Viper reference with field `val` (in Viper, all objects have all fields declared in the program).

the permissions are not available) and assigns non-deterministic values to the corresponding memory locations (to reflect that the environment could now modify them); `inhale` A analogously assumes constraints and adds permissions.

Regions and Assertions. TaDA's regions introduce various resources such as region predicates and guards. We encode these into Viper permissions and predicates as summarized in Fig. 4 (left). Each region `R` gives rise to a corresponding predicate, which is defined by the region interpretation. A region's abstract state may be accessed by a Viper function `R_State`, which is defined based on the region's `state` clause, and depends on the region predicate. Moreover, we introduce an abstract Viper predicate `R_g` for each guard `g` of the region.

These declarations allow us to encode most TaDA assertions in a fairly straightforward way. E.g. the assertion $\texttt{Lock}_r(\texttt{x}, s)$ from Fig. 1 is encoded as a combination of a region predicate and the function yielding its abstract state: `Lock(`r`,x) && Lock_State(`r`,x) ==` s. We encode region identifiers as references in Viper, which allows us to use the permissions and values of designated fields to represent resources and information associated with a region instance. E.g. we use the permission `acc(`r`.diamond)` to encode the TaDA resource $r \Rightarrow \blacklozenge$.

Rule Applications. Proof candidates are tree structures, where each premise of a rule application R is established as the conclusion of another rule application, as illustrated on the right. To check the validity of a candidate, we check the validity of each rule application. For rules that are natively supported by Viper (e.g. the assignment rule), Viper performs all necessary checks. Each other rule application is checked via an encoding into the following sequence of Viper instructions: (1) Exhale the precondition P_c of the conclusion to check that the required assertion holds. (2) Inhale the precondition P_p of the premise since it may be assumed when proving the premise. (3) After the code s of the premise, exhale the postcondition Q_p of the premise to check that it was established by the proof for the premise. (4) Inhale the postcondition Q_c of the conclusion. Steps 2 and 3 are performed for each premise of the rule. Moreover, we assert the side conditions of each rule. If a proof candidate is invalid, e.g. composes incompatible rules, one of the checks above fails and the candidate is rejected.

$$\dfrac{\dfrac{\vdots}{\{\ P_p\ \}\ s\ \{\ Q_p\ \}}}{\{\ P_c\ \}\ s\ \{\ Q_c\ \}}\ (R) \qquad \vdots$$

Using this encoding of rule applications as building blocks, we can assemble entire procedure proofs as follows: for each procedure, we inhale its precondition, encode the rule application for its body, and then exhale its postcondition.

Example: Stabilizing Assertions. Recall that an assertion A is stable if and only if the environment cannot invalidate A by performing any legal region updates. In practice, this means that the environment cannot hold a guard that allows it to change the state of a region in a way that violates A. The challenge of *checking* stability as a side-condition is to *avoid higher-order quantification* over region instances and guards, which is hard to automate. We address this challenge by eagerly *stabilizing* assertions in the Viper encoding, i.e. we weaken Viper's verification state such that the remaining information about the state is stable. We achieve this effect by first assigning non-deterministic values to the region state and then constraining these to be within the states permitted by the region's transition system, taking into account the guards the environment could hold. The Viper code for stabilizing instances of `Lock` can be found in the full paper [49].

7 Evaluation

We evaluated Voila on nine benchmark examples from Caper's test suite, with the Treiber's stack [41] variant `BagStack` being the most complex example, and report verification times and annotation overhead. Each example has been verified in two versions: a version with Caper's comparatively *weak* non-atomic specifications, and another version with TaDA's *strong* atomic specifications; see Sec. 8 for a more detailed comparison of Voila and Caper. An additional example, `CounterCl`, demonstrates the encoding of a custom guard algebra not supported in Caper (see the full paper [49]). To evaluate performance stability, we seeded four examples with errors in the loop invariant, procedure postcondition, code, and region specification, respectively. Our benchmark suite is relatively small, but each example involves nontrivial specifications. To the best of our knowledge, no other (semi-)automated tool is able to verify similarly strong specifications.

Program	LOC	Stg	Wk	Cpr
SLock	15	2.6	2.1	1.4
TLock	23	21.8	8.1	2.4
TLockCl	16	2.9	2.6	0.5
CASCtr	25	3.9	2.7	1.5
BoundedCtr	24	8.1	5.1	63.1
IncDecCtr	28	4.2	3.1	2.9
ForkJoin	16	2.1	1.3	1.0
ForkJoinCl	28	2.9	2.3	1.6
BagStack	29	29.9	18.0	211.6
CounterCl	45	-	5.8	-

Program	Err	Stg	Wk	Cpr
CASCtr	L	1.5	1.9	1.5
	P	2.5	1.9	11.2
	C	1.5	1.2	0.5
	R	1.2	1.1	0.3
TLock	L	3.9	7.2	2.0
	P	7.2	3.4	2.4
	C	15.6	1.8	0.6
	R	4.1	1.8	0.7
TLockCl	P	2.9	2.6	143.4
	C	2.5	2.5	115.5
	R	1.8	1.7	5.0
BagStack	L	26.5	17.8	> 600
	P	27.9	17.7	> 600
	C	26.3	17.8	> 600
	R	14.4	9.2	216.6

Fig. 5: Timings in seconds for successful (left table) and failing (right table) verification runs; lines of code (LOC) are given for Voila programs and exclude proof annotations. *Stg*/*Wk* denote strong/weak Voila specifications; *Cpr* abbreviates Caper. Programs include spin and ticket locks, counters (*Ctr*), and client programs (*Cl*) using the proven specifications. Errors (*Err*) were seeded in loop invariants (*L*), postconditions (*P*), code (*C*), and region specifications (*R*).

Performance. Fig. 5 shows the runtime for each example in seconds. All measurements were carried out on a Lenovo W540 with an Intel Core i7-4800MQ and 16GB of RAM, running Windows 10 x64 and Java HotSpot JVM 18.9 x64; Voila was compiled using Scala 2.12.7. We used a recent checkout of Viper and Z3 4.5.0 x64 (we failed to compile Caper against newer versions of Z3). Each example was verified ten times (on a continuously-running JVM); after removing the highest and lowest measurement, the remaining eight values were averaged. Caper (which compiles to native code) was measured analogously.

Overall, Voila's verification times are good; most examples verify in under five seconds. Voila is slower than Caper and its logic-specific symbolic execution engine, but it exhibits stable performance for successful and failing runs, which is crucial in the common case that proof outlines are developed interactively, such that the checker is run frequently on incorrect versions. As demonstrated by the error-seeded versions of `TLockCl` and `BagStack`, Caper's performance is less stable.

Another interesting observation is that strong specifications typically do not take significantly longer to verify, although only they require the full spectrum of TaDA ingredients and make use of TaDA's most complex rules, MakeAtomic and UpdateRegion. Notable exceptions are: `BagStack`, where only the strong specification requires sequence theory reasoning; and `TLock` and `BoundedCtr`, whose complex transition systems with many disjunctions significantly increase the workload when verifying atomicity rules such as MakeAtomic.

Automation. Voila's annotation overhead, averaged over the programs with *strong* specifications from Fig. 5, is 0.8 lines of proof annotations (not counting declarations and procedure specifications; neither for Caper) per line of code, which demonstrates the high degree of automation Voila achieves. Caper has an

average annotation overhead of 0.13 for its programs from Fig. 5, but significantly weaker specifications. Verifying only the latter in Voila does not reduce annotation overhead significantly since Voila was designed to support TaDA's strong specifications. The overhead reported for encodings into interactive theorem provers such as Coq [10,17,18,45] is typically much higher, ranging between 10 and 20.

8 Related Work

We compare Voila to three groups of tools: automated verifiers, focusing on automation; proof checkers, focusing on expressiveness; and proof outline checkers, designed to strike a balance between automation and expressiveness. Closest to our work in the kind of supported logic is the automated verifier Caper [8], from which we drew inspiration, e.g. for how to specify region transition systems. Caper supports an improved version of CAP [7], a predecessor logic of TaDA. Caper's symbolic execution engine achieves an impressive degree of automation, which, for more complex examples, is higher than Voila's. Caper's automation also covers slightly more guard algebras than Voila. However, the automation comes at the price of expressiveness, compared to Voila: postconditions are often significantly weaker because the logic does not support linearizability (or any other notion of abstract atomicity). E.g. Caper cannot prove that the spinlock's `unlock` procedure actually releases the lock. As was shown in Sec. 7, Caper is typically faster than Voila, but exhibits less stable performance when a program or its specifications are wrong.

Other automated verifiers for fine-grained concurrency reasoning are SmallfootRG [5], which can prove memory safety, but not functional correctness, and CAVE [44], which can prove linearizability, but cannot reason about non-linearizable code (which TaDA and Voila can). VerCors [26] combines a concurrent separation logic with process-algebraic specifications; special program annotations are used to relate concrete program operations to terms in the abstract process algebra model. Reasoning about the resulting term sequences is automated via model checking, but is non-modular. Summers et al. [39] present an automated verifier for the RSL family of logics [45,9,10] for reasoning about weak-memory concurrency. Their tool also encodes into Viper and requires very few annotations because proofs in the RSL logics are more stylized than in TaDA.

A variety of complex separation logics [45,23,43,37,9,10,12,19,16] are supported by proof checkers, typically via Coq encodings. As discussed in the introduction, such tools strike a different trade-off than proof outline checkers: they provide foundational proofs, but typically offer little automation, which hampers experimenting with logics.

Starling [46] is a proof outline checker and closest to Voila in terms of the overall design, but it focuses on proofs that are *easy* to automate. To achieve this, it uses a simple instantiation of the Views meta-logic [6] as its logic. Starling's logic does not enable the kind of strong, linearizability-based postconditions that Voila can prove (see the discussion of Caper above). Starling generates proof obligations that can be discharged by an SMT solver, or by GRASShopper [31] if the program

requires heap reasoning. The parts of an outline that involve the heap must be written in GRASShopper's input language. In contrast, Voila does not expose the underlying system, and users can work on the abstraction level of TaDA.

VeriFast [14] can be seen as an outline checker for a separation logic with impressive features such as higher-order functions and predicates. It has no dedicated support for fine-grained concurrency, but the developers manually encoded examples such as concurrent stacks and queues. VeriFast favors expressiveness over automation: proofs often require non-trivial specification adaptations and substantial amounts of ghost code, but the results typically verify quickly.

9 Conclusion

We introduced Voila, a novel proof outline checker that supports most of TaDA's features, and achieves a high degree of automation and good performance. This enables concise proof outlines with a strong resemblance of TaDA.

Voila is the first deductive verifier that can reason automatically about a procedure's effect at its linearization point, which is essential for a wide range of concurrent programs. Earlier work either proves much weaker properties (the preservation of basic data structure invariants rather than the functional behavior of procedures) or requires substantially more user input (entire proofs rather than concise outlines).

We believe that our systematic approach to developing Voila can be generalized to other complex logics. In particular, encoding proof outlines into an existing verification framework allows one to develop proof outline checkers efficiently, without developing custom proof search algorithms. Our work also illustrates that an intermediate verification language such as Viper is suitable for encoding a highly specialised program logic such as TaDA. During the development of Voila, we uncovered and fixed several soundness and modularity issues in TaDA, which the original authors acknowledged and had partly not been aware of. We view this as anecdotal evidence of the benefits of tool support that we described in the introduction.

Voila supports the vast majority of TaDA's features; most of the others can be supported with additional annotations. The main exception are TaDA's hybrid assertions, which combine atomic and non-atomic behavior. Adding support for those is future work. Other plans include an extension of the supported logic, e.g. to handle extensions of TaDA [36,11].

Acknowledgements. We thank the anonymous referees of this paper, and earlier versions thereof, for suggesting many improvements to the explanation of our work. We are also thankful to Thomas Dinsdale-Young and Pedro da Rocha Pinto for instructive discussions about their work, TaDA, and for feedback on Voila.

References

1. Apt, K.R., de Boer, F.S., Olderog, E.: Verification of Sequential and Concurrent Programs. Texts in Computer Science, Springer (2009)
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: FMCO. Lecture Notes in Computer Science, vol. 4111, pp. 115–137. Springer (2005)
3. Brookes, S., O'Hearn, P.W.: Concurrent separation logic. SIGLOG News **3**(3), 47–65 (2016)
4. Brookes, S.D.: A semantics for concurrent separation logic. In: CONCUR. Lecture Notes in Computer Science, vol. 3170, pp. 16–34. Springer (2004)
5. Calcagno, C., Parkinson, M.J., Vafeiadis, V.: Modular safety checking for fine-grained concurrency. In: SAS. Lecture Notes in Computer Science, vol. 4634, pp. 233–248. Springer (2007)
6. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M.J., Yang, H.: Views: compositional reasoning for concurrent programs. In: POPL. pp. 287–300. ACM (2013)
7. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: ECOOP. Lecture Notes in Computer Science, vol. 6183, pp. 504–528. Springer (2010)
8. Dinsdale-Young, T., da Rocha Pinto, P., Andersen, K.J., Birkedal, L.: Caper - Automatic verification for fine-grained concurrency. In: ESOP. Lecture Notes in Computer Science, vol. 10201, pp. 420–447. Springer (2017)
9. Doko, M., Vafeiadis, V.: A program logic for C11 memory fences. In: VMCAI. Lecture Notes in Computer Science, vol. 9583, pp. 413–430. Springer (2016)
10. Doko, M., Vafeiadis, V.: Tackling real-life relaxed concurrency with FSL++. In: ESOP. Lecture Notes in Computer Science, vol. 10201, pp. 448–475. Springer (2017)
11. D'Osualdo, E., Farzan, A., Gardner, P., Sutherland, J.: TaDA Live: Compositional reasoning for termination of fine-grained concurrent programs. CoRR **abs/1901.05750** (2019)
12. Frumin, D., Krebbers, R., Birkedal, L.: Reloc: A mechanised relational logic for fine-grained concurrency. In: LICS. pp. 442–451. ACM (2018)
13. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
14. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: NASA Formal Methods. Lecture Notes in Computer Science, vol. 6617, pp. 41–55. Springer (2011)
15. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress. pp. 321–332 (1983)
16. 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)
17. Kaiser, J., Dang, H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In: ECOOP. LIPIcs, vol. 74, pp. 17:1–17:29. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
18. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: SOSP. pp. 207–220. ACM (2009)

19. 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. PACMPL **2**(ICFP), 77:1–77:30 (2018)
20. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010)
21. Mooij, A.J., Wesselink, W.: Incremental verification of owicki/gries proof outlines using PVS. In: Lau, K., Banach, R. (eds.) International Conference on Formal Engineering Methods (ICFEM). Lecture Notes in Computer Science, vol. 3785, pp. 390–404. Springer (2005)
22. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: VMCAI. Lecture Notes in Computer Science, vol. 9583, pp. 41–62. Springer (2016)
23. Nanevski, A., Ley-Wild, R., Sergey, I., Delbianco, G.A.: Communicating state transition systems for fine-grained concurrent resources. In: ESOP. Lecture Notes in Computer Science, vol. 8410, pp. 290–310. Springer (2014)
24. O'Hearn, P.W.: Resources, concurrency and local reasoning. In: CONCUR. Lecture Notes in Computer Science, vol. 3170, pp. 49–67. Springer (2004)
25. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: CSL. Lecture Notes in Computer Science, vol. 2142, pp. 1–19. Springer (2001)
26. Oortwijn, W., Blom, S., Gurov, D., Huisman, M., Zaharieva-Stojanovski, M.: An abstraction technique for describing concurrent program behaviour. In: VSTTE. Lecture Notes in Computer Science, vol. 10712, pp. 191–209. Springer (2017)
27. Owicki, S.S.: Axiomatic Proof Techniques for Parallel Programs. Outstanding Dissertations in the Computer Sciences, Garland Publishing, New York (1975)
28. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inf. **6**, 319–340 (1976)
29. Parkinson, M.J., Summers, A.J.: The relationship between separation logic and implicit dynamic frames. Logical Methods in Computer Science **8**(3:01), 1–54 (2012)
30. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: POPL. pp. 247–258. ACM (2005)
31. Piskac, R., Wies, T., Zufferey, D.: GRASShopper - complete heap verification with mixed specifications. In: TACAS. Lecture Notes in Computer Science, vol. 8413, pp. 124–139. Springer (2014)
32. Raad, A., Villard, J., Gardner, P.: CoLoSL: Concurrent local subjective logic. In: Vitek, J. (ed.) ESOP. Lecture Notes in Computer Science, vol. 9032, pp. 710–735. Springer (2015)
33. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE Computer Society (2002)
34. da Rocha Pinto, P.: Reasoning with time and data abstractions. Ph.D. thesis, Imperial College London, UK (2016)
35. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A logic for time and data abstraction. In: ECOOP. Lecture Notes in Computer Science, vol. 8586, pp. 207–231. Springer (2014)
36. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P., Sutherland, J.: Modular termination verification for non-blocking concurrency. In: ESOP. Lecture Notes in Computer Science, vol. 9632, pp. 176–201. Springer (2016)
37. Sergey, I., Nanevski, A., Banerjee, A.: Mechanized verification of fine-grained concurrent programs. In: PLDI. pp. 77–87. ACM (2015)

38. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: ECOOP. Lecture Notes in Computer Science, vol. 5653, pp. 148–172. Springer (2009)
39. Summers, A.J., Müller, P.: Automating deductive verification for weak-memory programs. In: TACAS (1). Lecture Notes in Computer Science, vol. 10805, pp. 190–209. Springer (2018)
40. Svendsen, K., Birkedal, L.: Impredicative concurrent abstract predicates. In: Shao, Z. (ed.) European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 8410, pp. 149–168. Springer (2014)
41. Treiber, R.K.: Systems programming: Coping with parallelism. Tech. Rep. RJ 5118, IBM Almaden Research Center (1986)
42. Turon, A., Dreyer, D., Birkedal, L.: Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In: Morrisett, G., Uustalu, T. (eds.) International Conference on Functional Programming (ICFP). pp. 377–390. ACM (2013)
43. Turon, A., Vafeiadis, V., Dreyer, D.: GPS: navigating weak memory with ghosts, protocols, and separation. In: OOPSLA. pp. 691–707. ACM (2014)
44. Vafeiadis, V.: Automatically proving linearizability. In: CAV. Lecture Notes in Computer Science, vol. 6174, pp. 450–464. Springer (2010)
45. Vafeiadis, V., Narayan, C.: Relaxed separation logic: a program logic for C11 concurrency. In: OOPSLA. pp. 867–884. ACM (2013)
46. Windsor, M., Dodds, M., Simner, B., Parkinson, M.J.: Starling: Lightweight concurrency verification with views. In: CAV. Lecture Notes in Computer Science, vol. 10426, pp. 544–569. Springer (2017)
47. Wolf, F.A., Schwerhoff, M., Müller, P.: The Voila source repository, `https://github.com/viperproject/voila`
48. Wolf, F.A., Schwerhoff, M., Müller, P.: Concise outlines for a complex logic: A proof outline checker for TaDA (2021). https://doi.org/10.5281/zenodo.5137791
49. Wolf, F.A., Schwerhoff, M., Müller, P.: Concise outlines for a complex logic: A proof outline checker for TaDA (full paper). CoRR **abs/2010.07080** (2020), `https://arxiv.org/abs/2010.07080`