Formal Methods in System Design
https://doi.org/10.1007/s10703-023-00427-w

ORIGINAL ARTICLE

Concise outlines for a complex logic: a proof outline checker for TaDA

Felix A. Wolf[1] · **Malte Schwerhoff**[1] · **Peter Müller**[1]

Received: 16 April 2022 / Accepted: 16 April 2023

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.

Keywords Software verification · Program verifier · Automated verification · Separation logic · Fine-grained concurrency · Formal methods

1 Introduction

Standard separation logic enables the modular verification of heap-manipulating sequential [1, 2] and data-race free concurrent programs [3, 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 [5], and rely-guarantee [6].

✉ Felix A. Wolf
felix.wolf@inf.ethz.ch

Malte Schwerhoff
malte.schwerhoff@inf.ethz.ch

Peter Müller
peter.mueller@inf.ethz.ch

[1] Department of Computer Science, ETH Zurich, Zurich, Switzerland

Published online: 31 July 2023

Examples include CAP [7], iCAP [8], CaReSL [9], CoLoSL [10], FCSL [11], GPS [12], RSL [13], and TaDA [14] (see Brookes et al. [15] 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 [13] and FCSL [11]. 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 [16] and Grasshopper [17] for traditional separation logics, and Caper [18] 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: ultimately based on a language semantics, with a minimal trusted core. 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 (which increase the trusted core).

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* [19, 20] (see, e.g., Pierce et al. [21] for a detailed discussion): 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 [22] and VeriFast [23] 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 this paper, we present Voila, a proof outline checker for TaDA [14], which goes beyond existing proof outline checkers and automated verifiers by supporting a substantially more complex program logic, which handles 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 [14, 24].

- A systematic approach to automate the expansion of a proof outline into a full *proof candidate* via a normal form and heuristics. Our approach automates most proof steps (e.g., 20 out of 22 for the running example from Fig. 1).
- An encoding of the proof candidate into Viper [25], which checks its validity without requiring any TaDA-specific proof search algorithms.
- The Voila proof outline checker, the *first* tool that supports specification for linearization points, provides a high degree of automation, and achieves good performance. Our submission artifact [26] contains the executable Voila tool; the Voila source code is also available [27].

Outline Sec. 2 gives an overview of the TaDA logic and illustrates our approach. Section 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. Sec. 7 provides a detailed soundness argument. In Sec. 8, we evaluate our technique by verifying several challenging examples. We discuss related work in Sec. 9 and conclude in Sec. 10.

Our full paper [28] 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; and the encoding of a custom guard algebra.

This paper is an extended version of a paper published at Formal Methods 2021 [29]. It has been revised to improve accessibility, particularly in Sec. 2, and extended with the soundness argument in Sec. 7.

2 Running example and TaDA overview

Figure 1 shows the first half of our running example, adapted from the original TaDA publication [14]: the TaDA proof outline of the `lock` procedure of a spinlock, whose atomic specifications capture its essence as a primitive for mutual exclusion. In Sec. 2.4, we then discuss a non-atomic specification derived from `lock` that conceptually ties a lock to an invariant. As in the original publication [14], the outline in Fig. 1 shows only two out of 22 proof steps, and omits most side conditions. In a TaDA proof outline, a proof step corresponds to the application of a TaDA rule, including suitable pre- and postconditions. Our outline shows applications of the rules MAKEATOMIC and UPDATEREGION. Deriving the shown pre- and postconditions may require additional rule applications, which are omitted.

We use our running example to introduce the necessary TaDA background, explain TaDA proof outlines, and illustrate the corresponding Voila proof outlines.

2.1 Regions and atomicity

TaDA targets shared-memory concurrency with sequentially-consistent memory, and TaDA programs manipulate *shared regions*: data structures that are concurrently modified according to a specified *protocol* (as in rely-guarantee reasoning [6]). A shared region such as $\text{Lock}_r(x, s)$ (subscript r uniquely identifies a specific region instance) is an abstraction over the region's content, analogous to abstract predicates [30] in traditional separation logic. The interpretation $I(\text{Lock}_r(x, s))$ defines the region's content. 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

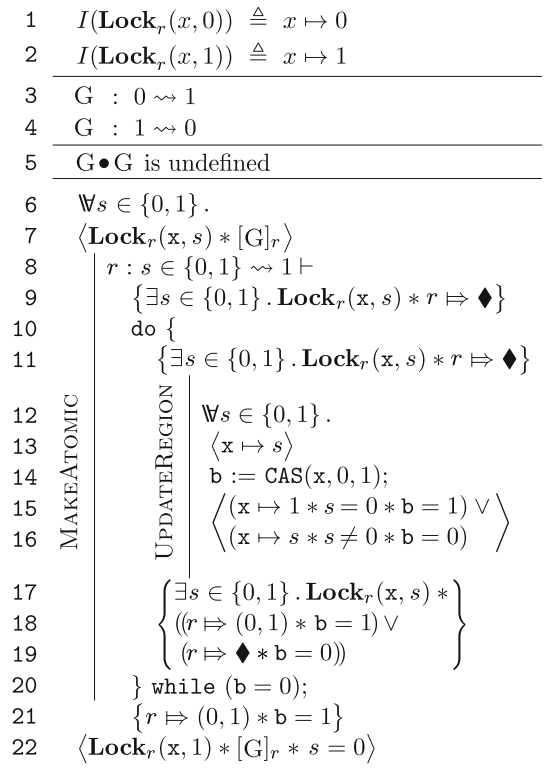

Fig. 1 First half of our running example: a spinlock with atomic TaDA specifications and shared region Lock; adapted with only minor changes from TaDA [14]. The lock region (lines 1–2) comprises a single memory location, whose value is either 0 (available) or 1 (acquired). Guard G allows locking and unlocking (lines 3–4), and is unique (line 5). The proof outline (lines 6–22) shows the implementation of a CAS-based Lock operation with atomic specifications. Levels (denoted by λ in TaDA) are omitted from the discussion in this paper, but supported by Voila and included in the full paper [28]

state (second region parameter) and the content of the memory location (value pointed to by x) coincide, but they may differ in general.

Unlike traditional abstract predicates, shared regions are *duplicable*, i.e., the equivalence $\mathrm{Lock}(x,s) \Leftrightarrow \mathrm{Lock}(x,s) * \mathrm{Lock}(x,s)$ holds. This allows multiple threads to obtain an instance of a Lock region, and to compete for the corresponding lock. However, note that duplicating a shared region indirectly also allows duplicating points-to predicates, which are unique in traditional separation logic and elsewhere in TaDA. This is nevertheless sound because TaDA's intricate proof rules ensure that a shared region is opened only for an abstractly-atomic duration, and that no two instances of the same region are opened simultaneously.

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). Using a unique guard in a context where multiple threads compete for acquiring a lock may seem counterintuitive, but the combination of unique guards and duplicable shared region assertions resolves this perceived conflict, as discussed in Sec. 2.4. Note that the transition system is defined relative

to a region's abstract value, not its internal memory values, which is not directly apparent in this example, since abstract and concrete values coincide.

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)

$$\mathbb{V}s \in \{0, 1\} \cdot \langle \text{Lock}_r(\text{x}, s) * [\text{G}]_r \rangle \ \texttt{lock}(\text{x}) \ \langle \text{Lock}_r(\text{x}, 1) * [\text{G}]_r * s = 0 \rangle$$

Atomic triples (angle brackets) express that their statement is linearizable [31]. 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). In contrast, pre- and postconditions of standard Hoare triples (curly braces) are interpreted as usual: relative to the start and the end of the specified statement's execution. Intuitively, it is the combination of linearizability, shared regions with abstract state, and guarded transition systems that establishes TaDA's *abstract atomicity*: operations (e.g., `lock`) appear atomic when interacted with on the level of a shared region (e.g., Lock(x,s)), and TaDA's derivation rules ensure that the abstraction holds, even if the underlying memory is manipulated non-atomically.

The *interference context* $\mathbb{V}s \in \{0, 1\}$ is a special binding for the abstract region state that forces callers of `lock` to guarantee that the environment keeps the lock state in the set $\{0, 1\}$ until the linearization point is reached. Correspondingly, it also requires the callees to not take the region out of this abstract state (i.e., $\{0, 1\}$) before its linearization point is reached. In this case, both restrictions are vacuous; in general, the interference context can be understood as a symmetric rely-guarantee condition.

The precondition of the triple states that an instance of guard G for region r, $[\text{G}]_r$, 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 (Lock(x,1)). Such precise specifications of state updates are enabled by the atomic triple's interpretation relative to the linearization point. In contrast, standard Hoare triples would have to account for potential environment interference *before* and *after* the linearization point—and can thus often only specify preservation of data structure invariants. In this paper, we refer to standard Hoare triples also as *non-atomic triples*.

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. The inner rule application, UpdateRegion, identifies the linearization point inside an abstractly-atomic code block. The surrounding rule application, MakeAtomic, then checks that there is exactly one such linearization point, ensuring that the block of code is indeed atomic w.r.t. a shared region abstraction, and establishes an atomic triple. This justifies the change from non-atomic premise triple (lines 9 and 21) to an atomic conclusion triple (lines 7 and 22), around the body of the CAS-based implementation of the `lock` procedure.

Rule MakeAtomic requires that the *atomicity context* of the premise triple, a set $\mathcal{A}$ of *pending updates* (for brevity, omitted from the previously-shown triple for `lock`(x)), includes any region updates performed by the statement of the triple. By tracking pending updates and allowing at most one per region ($r \notin \mathcal{A}$ in Fig. 2), MakeAtomic intuitively prevents more

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

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

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

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. $\mathbf{R}_r(\vec{z},x)$ and $I(\mathbf{R}_r(\vec{z},x))$ are the shared region and its content, respectively. 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. $P(x)$, $Q_1(x, y, w)$, and $Q_2(x, w)$ are some TaDA assertions. USEATOMIC is a special combination of MAKEATOMIC and UPDATEREGION, where the linearizable statements itself is the linearization point. Again, $P(x)$ and $Q(x, y)$ are some TaDA assertions

than one observable change to the same shared region from happening during an abstractly-atomic operation.

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 following the original TaDA publication, 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* $r \Mapsto (0, 1)$, which indicates that the region was updated from abstract state 0 to 1. Intuitively, the witness resource guarantees that there is exactly one linearization point where the relevant state update happened. This guarantee enables MAKEATOMIC to establish atomic triples from non-atomic triples. Furthermore, the witness resource is needed to carry sufficient information from the linearization point (which may not be the last statement in the procedure) to the point at which the operation's postcondition is to be established: the latter is interpreted w.r.t. the linearization point, but other threads may have changed the shared region since then. Finally, 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.

Note that the proof outline also illustrates how to convert between atomic and non-atomic triples in TaDA. The MAKEATOMIC rule is the only rule that can establish atomic triples, justified by the single linearization point. Conversely, an atomic triple can always be converted to a non-atomic triple by weakening its postcondition to account for the environment's interference. In Fig. 1, this happens around UPDATEREGION. The need for weakening postconditions is discussed in more detail in Sec. 5, in the context of stable assertions.

2.3 Voila proof outline

Figure 3 shows the *complete* proof outline of our example discussed so far, 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. The successful verification shows 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.

2.4 Locks with resource invariants

This subsection completes our running example by showing a TaDA outline (Fig. 4) and corresponding Voila code (Fig. 5) for a specification of `lock` that ties the spinlock to a resource invariant the lock protects. The resources in this invariant are then temporarily transferred to any threads that acquires the lock.

Following the TaDA publication, we introduce a new shared region called CAPLock (lines 1–4 of Fig. 4), which wraps an instance of the previously declared Lock region, and two new guards (lines 5–6): a vacuous *empty* guard **0** that is always available and used to acquire a lock, and a unique guard U for releasing it. Note that **0** and U both guard transitions of the abstract state of the CAPLock region. Transitioning the underlying Lock region (i.e., actually acquiring and releasing the lock) still requires the previously-introduced guard G. Consequently, an unlocked CAPLock region (lines 1–2) contains guards G and U, and resource invariant *Inv* (left abstract for brevity). In contrast, when locked (lines 3–4), both resources U and *Inv* are owned by the lock holder, and the shared region only contains guard G.

The body of the proof outline (lines 8–15) uses the previously established atomic TaDA triple for procedure `lock` to derive the following, non-atomic TaDA triple:

$$\mathcal{A} \vdash \{\exists v \in \{0,1\} \cdot \text{CAPLock}_a(r, \text{x}, v)\}\texttt{lock(x)}\{\text{CAPLock}_a(r, \text{x}, 1) * [\text{U}]_a * \mathit{Inv}\}$$

As before, the proof outline omits most steps, and shows only two rule applications: the frame rule and USEATOMIC. The application of the frame rule enables us to preserve the assertion $v = 0 \rightarrow [\text{U}]_a * \mathit{Inv}$ (abbreviated as F in the figure) across the call to `lock`. For the postcondition, we use an omitted rule of consequence to derive from $v = 0$ and

```
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`. Operator `&&` denotes separating conjunction

$v = 0 \rightarrow [\mathrm{U}]_a * \mathit{Inv}$ the assertion $[\mathrm{U}]_a * \mathit{Inv}$. Intuitively, the application of UseAtomic (also shown in Fig. 2) applies MakeAtomic and UpdateRegion together at once.

The complete Voila proof outline for CAPLock is shown in Fig. 5, where procedure caplock has the desired specification. The example verifies in Voila when combined with the previously shown code from Fig. 3.

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 [19], or tools that automatically check the validity of a proof outline [22, 23, 32]; 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

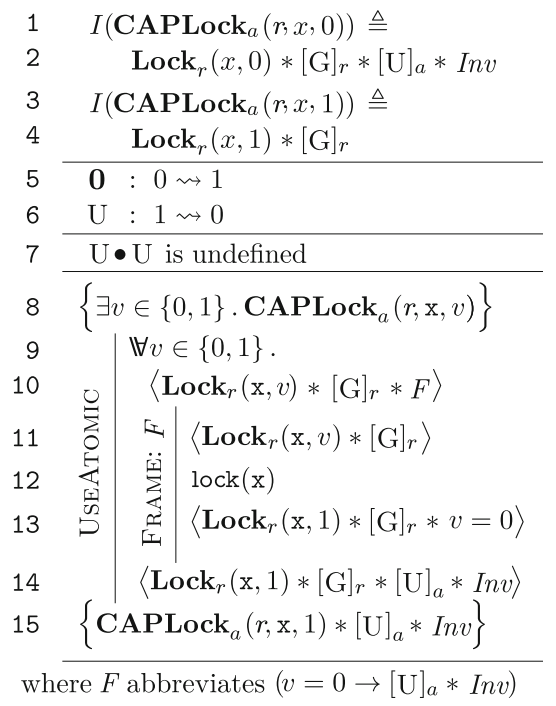

Fig. 4 TaDA declarations and proof outline for a shared region CAPLock, taken (with minor changes) from the TaDA publication [14], and building on the lock example from Fig. 1. The additional declarations and the outline's postcondition provide the usual semantics of a lock that protects a resource invariant: the vacuous empty guard **0** allows arbitrarily many clients to compete for the lock, but only the holder of the unique guard U can release the lock. Lock holders also temporarily gain ownership of the lock's resource invariant. Levels are again omitted, but supported by Voila and included in the full paper [28]

```
predicate Inv() /* Invariant, left abstract */

region CAPLock(id a, id r, cell x)
  interpretation {
    Lock(r, x, ?v) && G@r && (v == 0 || v == 1) && (v == 0 ==> U@a && Inv())
  }
  state { v }
  guards { duplicable Z; unique U; }
  actions { Z: 0 ~> 1; U: 1 ~> 0; }

procedure caplock(id a, id r, cell x)
  requires CAPLock(a, r, x) && Z@a;
  ensures  CAPLock(a, r, x, 1) && U@a && Inv();
{
  use_atomic using CAPLock(a, r, x) with Z@a {
    lock(r, x);
  }
}
```

Fig. 5 The Voila proof outline of TaDA's CAPLock, building on our lock example from Fig. 3, and strongly resembling the TaDA proof outline from Fig. 4. Note that Voila does not yet support TaDA's empty guard; instead, we use a duplicable guard Z

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. (3) The effort of authoring a proof outline can be greatly reduced by applying some rules heuristically, based on information 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 [33] applies heuristics to guess termination measures for loops, but also offers an annotation to provide a measure manually, if necessary. Another common example is the rule of consequence: SMT-based verifiers (such as Voila) automatically discharge most entailment checks, but may require additional user annotations in cases where the underlying SMT solver is incomplete.

The rule classification depends on the proof search capabilities of the verification tool that is used to check the proof outline. We use Viper [25], 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 [28], 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 [18]); additional ones can be encoded, see the full paper [28]. 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 $\forall s \in \{0, 1\}$.

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 [14, 24] 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* [24, 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. 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 *all* 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. Out of the 22 rule applications for our running example, our heuristics infer 17 applications of bridge rules. Three syntax-driven rules are also applied automatically, such that only two key rules require manual annotations. The complete TaDA proof shown in App. A details all inferred applications of bridge rules and syntax-driven rules.

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 form is established by a combination of syntactic checks and proof obligations in the final Viper encoding. Its main restrictions are as follows: (1) A triple is atomic if and only if the enclosed Voila outline statement is abstract atomic, namely a CAS operation, a call to an abstract atomic procedure, or a key rule statement. As a consequence, we can 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 [24] 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 ($\forall$), 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: (1) to determine when to change triple atomicity, (2) to ensure stable frames by construction, (3) to compute atomicity context ranges, (4) to compute levels, and (5) 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 [28] 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 prove 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.

For our running example from Fig. 1, four of our heuristics are necessary to complete the proof candidate. The heuristic (1) is necessary around UpdateRegion to change the triple atomicity. The heuristic (2) is necessary around the CAS operation to frame information about the arguments. The heuristic (3) is necessary so that clients can call the Lock procedure. Lastly, the heuristic (4) is necessary around UpdateRegion to change levels.

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 [25].

```
⟦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 OR 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. 6 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)

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 [35, 36] 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 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 conversely 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. 6 (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 $\text{Lock}_r(x, s)$ 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 $r \Rightarrow \blacklozenge$.

$$\dfrac{\dfrac{\vdots}{\{\ P_p\ \}\ s_p\ \{\ Q_p\ \}}}{\{\ P_c\ \}\ s_c\ \{\ Q_c\ \}}\ (R)$$
$$\vdots$$

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 above. In TaDA, the statement of the premise s_p is guaranteed to be a substatement of the statement of the conclusion s_c. 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 encoding of the proof for 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.

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 [28].

7 Soundness

Voila is sound if the successful verification of a procedure in Voila implies that the procedure's specification can be derived in TaDA. That is, our soundness argument builds on the soundness of the TaDA logic itself, which has been proven separately [37] w.r.t. an operational semantics. Voila succeeds if the encoded proof candidate of the procedure successfully verifies in Viper. Consequently, to show soundness of Voila, we need to show that successful verification of a proof candidate in Viper implies the existence of a corresponding TaDA proof for the given procedure and its specification.

Notations. Before we formalize our argument, we introduce basic terminology and notation. As discussed in Sec. 6, proof candidates are derivation trees in the TaDA logic. We refer to the proof candidate that is inferred for a Voila outline statement s as the proof candidate for s. We introduce a function C to model the inference of proof candidates, where $C(s)$ is the proof candidate of the Voila outline statement s. The entire proof candidate for a procedure

is obtained by applying C to the procedure's body. For a proof candidate p, p's *root* is the last rule application of p, which derives the overall conclusion, and p's *children* are the proof candidates whose roots are rule applications to the premises of that last rule application in p.

As discussed in Sec. 6, proof candidates are encoded to Viper statements by encoding all rule applications of the proof candidate. For a proof candidate p, $[\![p]\!]$ denotes the Viper statement that p is encoded to. We use the same notation to encode TaDA assertions and expressions. *Viper verification states* model the entire knowledge of the Viper verifier at a specific point in a Viper program. Technically, each Viper verification state is a set of concrete Viper states; this set can be characterized by a Viper assertion. For a Viper assertion A, $\Upsilon(A)$ denotes the initial Viper verification state after inhaling A only. The expression $\mathsf{post}(c, \upsilon)$ denotes the Viper verification state resulting from verifying a Viper statement c starting from the Viper verification state υ. The function post is analogous to the strongest postcondition of standard Hoare logic. We refer to υ and $\mathsf{post}(c, \upsilon)$ as the Viper verification pre- and poststate of c, respectively. The special error Viper verification state $\lightning$ models that an assertion failed during verification. We use s, p, υ to range over Voila outline statements, proof candidates, and Viper verification states, respectively.

7.1 Proof overview

We split our soundness argument into five steps: (1) We determine invariants on the Viper verification pre- and poststates of encoded rule applications. (2) We define a *judgment mapping* $(\!| \upsilon, p |\!)$, which takes a Viper verification state υ satisfying our invariants from the first step, together with a proof candidate p, and returns a TaDA judgment. We refer to $(\!| \upsilon, p |\!)$ also as the TaDA judgment of υ and p, sometimes omitting υ.

This judgment mapping establishes our connection between verifying a Viper program and deriving a TaDA proof. (3) We show that the Viper encoding of single rule applications is sound: Consider a proof candidate p whose root is a rule application for a TaDA rule r. Let the proof candidates $p'_1, \ldots, p'_N$ be p's children and let υ_0 be a Viper verification state satisfying our invariants. Since soundness considers only Voila statements, for which the Viper encoding verifies successfully, we may assume that no assertion fails when verifying the Viper encoding of r's rule application starting from υ_0.

$$\frac{(\!| \upsilon_1, p'_1 |\!) \quad \ldots \quad (\!| \upsilon_N, p'_N |\!)}{(\!| \upsilon_0, p |\!)}\,(r)$$

As illustrated above, we then show that the TaDA rule r can be applied correctly to derive p's TaDA judgment $(\!| \upsilon_0, p |\!)$ as the conclusion. The rule application must contain the TaDA judgments $(\!| \upsilon_1, p'_1 |\!), \ldots, (\!| \upsilon_N, p'_N |\!)$ of p's children as the premises. The Viper verification states $\upsilon_1, \ldots, \upsilon_N$ are the prestates of the childrens' Viper encodings. (4) We show inductively that the Viper encoding of proof candidates is sound: Let s be a Voila outline statement and υ be a Viper verification state satisfying our invariants. We may assume that no assertion failed when verifying the encoded proof candidate $[\![C(s)]\!]$ of s starting from υ. We then show that the TaDA judgment $(\!| \upsilon, C(s) |\!)$ of the proof candidate $C(s)$ is derivable in TaDA. (5) We show soundness of the specification encoding: Consider a successfully-verified Voila procedure with precondition P, postcondition Q, and body s. We show that the procedure's specification can be derived in TaDA as a conclusion using the rule of consequence on the TaDA judgment $(\!| \Upsilon([\![P]\!]), C(s) |\!)$ of the Viper verification state $\Upsilon([\![P]\!])$ from the encoded precondition $[\![P]\!]$ and of the procedure's proof candidate $C(s)$.

Combining these steps, we formally connect verification of an encoded proof candidate to derivability of a corresponding TaDA proof, resulting in the soundness of Voila. We first illustrate this approach in more detail on a simplified version of TaDA. Afterwards, we discuss how we apply this approach to full TaDA.

7.2 Proof for simplified TaDA

For a simplified version of TaDA, assume that TaDA judgments are standard Hoare judgments of the form $\vdash \{P\}\,\hat{s}\,\{Q\}$. We omit atomic triples, levels, atomicity contexts, interference contexts, and the requirement that pre- and postconditions are stable. These features are discussed in Sec. 7.3. Furthermore, for simplicity, we do not distinguish between Voila and TaDA assertions since they differ only in syntax. For our simplified version of TaDA, we illustrate how to instantiate our five aforementioned steps to show soundness.

Step 1: Invariants. To prove soundness, we will later (in step 2) connect Viper verification states to TaDA pre- and postconditions. However, not every Viper verification state can be connected to a TaDA assertion. E.g., TaDA does not support fractional permissions [38] for points-to predicates ($\texttt{x.f} \mapsto v$), whereas fractional permissions are generally possible in Viper verification states. We define invariants on Viper verification states that rule out Viper verification states that do not correspond to TaDA assertions. These invariants have to hold only for those Viper verification states that we have to connect to TaDA assertions, namely the pre- and poststates of encoded rule applications. In particular, intermediate Viper verification states of our Viper encoding do not have to satisfy the invariants. We use these invariants in the definition of the judgment mapping.

Step 2: Judgment Mapping. For our simplified version of TaDA, we define a judgment mapping for a Viper verification state v and the proof candidate of a Voila outline statement s as $(\!|\, v, C(s) \,|\!) = \{\phi(v)\}\llcorner s \lrcorner\{\phi(\mathsf{post}([\![\, C(s) \,]\!], v))\}$, where ϕ maps Viper verification states to TaDA assertions and $\llcorner\cdot\lrcorner$ maps Voila outline statements to TaDA statements. We refer to ϕ as the *TaDA assertion interpretation*. The definition of ϕ is very technical and omitted for the sake of simplicity. Conversely, the definition of $\llcorner\cdot\lrcorner$ is straightforward. The function removes all rule annotations from a Voila outline statement, e.g., $\llcorner$`update_region{b:= CAS(x,0,1)}`$\lrcorner$ `is b:= CAS(x,0,1)`. As explained for step 1, only Viper verification pre- and poststates of encoded rule applications are connected to TaDA assertions. To use the judgment mapping, we have to show that the invariants hold for the verification state-argument (v above).

To lift deductions at the level of Viper verification states to deductions at the level of TaDA assertions, we prove that the TaDA assertion interpretation ϕ satisfies two properties: (1) An entailment in Viper, e.g., $v_1 \vDash_{\mathsf{Viper}} v_2$ for some Viper verification states v_1 and v_2 (recall that Viper verification states correspond to Viper assertions), implies the corresponding entailment in TaDA, i.e., $\phi(v_1) \vDash_{\mathsf{TaDA}} \phi(v_1)$. (2) ϕ is the inverse of the encoding, i.e., $\phi(\Upsilon([\![\, P \,]\!])) = P$ for all TaDA assertions P. Using these two properties, we can show that if an encoded TaDA assertion $[\![\, P \,]\!]$ is successfully verified starting from a Viper verification state v, then $\phi(v)$ entails P in TaDA, i.e., $\phi(v) \vDash_{\mathsf{TaDA}} P$ holds. This is the basis for checking the correctness of rule applications in Viper.

Step 3: Single Rule Applications. Consider a proof candidate p whose root is an application of the syntax-driven rule for loops. Let the TaDA statement that the rule is applied to be `do invariant` I `{`$\hat{s}$`} while(`b`)` for some TaDA expression b, TaDA statement $\hat{s}$, and TaDA invariant I. Viper has while loops, but not do-while loops. A simplified Viper encoding

of the loop rule application is thus $[\![p']\!]$; `while`$([\![b]\!])$ `invariant` $[\![I]\!]$ `{`$[\![p']\!]$`}`, where the proof candidate p' is the child of p. If verification is successful, then the TaDA invariant I is preserved after the first execution of the loop body. Thus, TaDA's do-while rule can be applied to the TaDA judgment $(\!| v, p |\!)$, where v is the Viper verification state the verification started from. We show soundness of the Viper encoding of single rule applications separately for each rule, obtaining a soundness lemma per rule.

Side conditions already guaranteed by neighboring rule applications, either through checking or by construction, are not checked again. Therefore, for some rules, some of their side conditions are established by the encoding of neighboring rule applications. For the rules where neighboring rule applications guarantee some side conditions, we obtain weaker soundness lemmas, requiring that the necessary side conditions already hold. These dependencies make induction over the proof candidates difficult. Thus, instead of induction over all proof candidates, we perform induction over the Voila outline statements, as shown in the next step.

Step 4: Proof Candidates. The following lemma (SE) formalizes soundness of the Viper encoding of proof candidates, as well as that our invariants on Viper verification states, referred to as $\mathbb{I}$, are maintained by the Viper encoding.

$$v \in \mathbb{I} \;\wedge\; \mathsf{post}([\![C(s)]\!], v) \neq \lightning \implies (\!| v, C(s) |\!) \;\wedge\; \mathsf{post}([\![C(s)]\!], v) \in \mathbb{I} \qquad \text{(SE)}$$

Verbally, the lemma expresses *"For all Voila outline statements s and Viper verification states satisfying our invariants $v \in \mathbb{I}$, the absence of failed assertions during the verification of the encoded proof candidate, written* $\mathsf{post}([\![C(s)]\!], v) \neq \lightning$*, implies that the proof candidate's TaDA judgment $(\!| v, C(s) |\!)$ is derivable in TaDA and that the Viper verification poststate satisfies our Viper verification state invariants"*. We apply the lemma to the body of a Voila procedure to get that the Voila proof candidate corresponds to a correct TaDA proof.

We prove the lemma (SE) by structural induction over Voila outline statements. In general, the induction proceeds as follows: Consider a compound outline statement $s\{s'\}$ (s is the compound, e.g., `update_region`, and s' is its body, e.g., `CAS(...)`) with a Viper encoding $[\![C(s\{s'\})]\!] = c_1; [\![C(s')]\!]; c_2$, where c_1 and c_2 are the Viper statements before and after the Viper encoding of the body's proof candidate, respectively. There are four Viper verification states of interest: the prestate v_0 of the compound statement s, the prestate $v_1 = \mathsf{post}(c_1, v_0,)$ of the body s', the poststate $v_2 = \mathsf{post}([\![s']\!], v_1,)$ of s', and the poststate $v_3 = \mathsf{post}(c_2, v_2,)$ of s. From the induction hypothesis, we get that the TaDA judgment $(\!| v_1, C(s') |\!) = \{\phi(v_1)\} \llcorner s' \lrcorner \{\phi(v_2)\}$ of the body's proof candidate is derivable in TaDA. We have to show that the TaDA judgment $(\!| v_0, C(s\{s'\}) |\!) = \{\phi(v_0)\} \llcorner s\{s'\} \lrcorner \{\phi(v_3)\}$ of the compound statement's proof candidate is derivable in TaDA. Showing this derivation corresponds to applying rules from TaDA to identify the missing proof steps (indicated by ?) in the TaDA proof below.

$$\dfrac{\dfrac{\vdots}{\{\phi(v_1)\} \llcorner s' \lrcorner \{\phi(v_2)\}}\,(\text{IH})}{\{\phi(v_0)\} \llcorner s\{s'\} \lrcorner \{\phi(v_3)\}}\,(?)$$

The application of IH denotes using the induction hypothesis that $\{\phi(v_1)\} \llcorner s' \lrcorner \{\phi(v_2)\}$ is derivable in TaDA. The necessary rule applications for the missing proof steps are determined by the proof candidate. For each rule application from the proof candidate, we apply the

lemma obtained from the soundness of the Viper encoding of single rule applications (step 3), ultimately closing the gap.

Step 5: Specification. The previous proof steps and, in particular, Lemma (SE) show the existence of a TaDA proof for a given Voila statement, but do not yet show that this TaDA proof also establishes the pre- and postcondition of the Voila statement, which we do next. Consider a Voila procedure with a precondition P, postcondition Q, and body s. We need to prove that the procedure's TaDA specification $\{P\} \llcorner s \lrcorner \{Q\}$ is derivable as a conclusion using the rule of consequence on the TaDA judgment $(\!| \Upsilon([\![P]\!]), C(s) |\!)$ of the procedure's proof candidate $C(s)$.

For our simplified version of TaDA discussed in this subsection, this property holds if (1) P entails $\phi(\Upsilon([\![P]\!]))$ and (2) Q is entailed by the TaDA assertion interpretation $\phi(\upsilon_{\mathsf{post}})$, where υ_{post} is the Viper verification poststate of the encoded proof candidate $[\![C(s)]\!]$. Both entailments follow from the properties we proved for the TaDA assertion interpretation ϕ, namely that ϕ is the inverse of the encoding and that ϕ lifts Viper entailments to TaDA entailments. For (2), it is relevant that the Viper encoding asserts $[\![Q]\!]$ directly after the encoded proof candidate.

Overall Soundness of Voila. The soundness of the Viper encoding of both, specification and proof candidate, enables us to show that Voila is sound. Again, consider a Voila procedure with a precondition P, postcondition Q, and body s. Verification succeeds, if Viper successfully verifies the encoded proof candidate. More concretely, for the Viper precondition $[\![P]\!]$, the Viper tool verifies the Viper statement $[\![C(s)]\!]$ without failing an assertion and verifies the Viper postcondition $[\![Q]\!]$ afterwards. For soundness, we have to show that the corresponding TaDA specification, namely $\{P\} \llcorner s \lrcorner \{Q\}$, is derivable in TaDA.

By instantiating v with $\Upsilon([\![P]\!])$, Lemma (SE) gives us that the TaDA judgment $(\!| \Upsilon([\![P]\!]), C(s) |\!)$ of the procedure's proof candidate is derivable in TaDA. From the specification encoding soundness (step 5), we get that the procedure's TaDA judgment $\vdash \{P\} \llcorner s \lrcorner \{Q\}$ can be derived from the TaDA judgment of the procedure's proof candidate using TaDA's rule of consequence. Combining both implies that the TaDA judgment $\{P\} \llcorner s \lrcorner \{Q\}$ is derivable in TaDA, completing the soundness argument.

As a technical detail, to apply lemma (SE), we need to guarantee that the initial Viper verification state $\Upsilon([\![P]\!])$ from the encoded precondition satisfies the invariants $\mathbb{I}$. For our simplified version of TaDA discussed in this subsection, the invariants are guaranteed by the syntactic restrictions of the Voila specification language. For full TaDA, syntactic restrictions do not suffice, since assertions also have to be stable. Voila verifies that user-provided assertions are well-defined, i.e., that their corresponding Viper states are contained in $\mathbb{I}$, using additional Viper proof obligations.

7.3 Generalization to TaDA

The proof sketch shown in Sec. 7.2 does not account for TaDA's atomic triples, stability requirements, and judgment parameters, namely level, atomicity context, and interference context. To generalize the proof to full TaDA, we introduce three extensions.

First, the semantics of a TaDA assertion differs depending on its use, i.e., whether it is a pre- or postcondition and whether it is part of an atomic or non-atomic TaDA triple. Therefore, we need multiple mappings from Viper verification states to TaDA assertions (previously, just ϕ). Second, we extend lemma (SE) according to point (2) of our normal form from Sec. 5. More concretely, we add to the invariants $\mathbb{I}$ the restrictions enforced by our normal form about when

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. 7 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*)

TaDA pre- and postconditions have to be stable. Lastly, Voila proves TaDA judgments for a set of parameters (point (4) of our normal form from Sec. 5). As a consequence, the judgment mapping changes. E.g., for a non-atomic Voila outline statement s, our extended judgment mapping has the shape $(\!|\, \upsilon, C(s) \,|\!) = \forall \lambda \in \mathbb{L}(\upsilon), \mathcal{A} \in \mathbb{A}(\upsilon).\ \lambda, \mathcal{A} \vdash \{\phi(\upsilon)\} \llcorner s \lrcorner \{\phi(\upsilon')\}$ where $\mathbb{L}(\upsilon)$ and $\mathbb{A}(\upsilon)$ are the set of levels and atomicity contexts that the TaDA judgment is proved for, respectively. For atomic triples, we use a set of interference contexts as well.

Soundness for full TaDA is proved as described in Sec. 7.2. Our full paper [28] demonstrates the four particularly challenging cases, namely calls, the change from an atomic TaDA judgment to a non-atomic TaDA judgment, `make_atomic`, and `update_region`.

8 Evaluation

We evaluated Voila on nine benchmark examples from Caper's test suite, with the Treiber's stack [39] 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. 9 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 [28]). To evaluate the performance for both successful and failing verification attempts, 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.

Performance. Fig. 7 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. 7, 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. 7, 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 [13, 40–42] is typically much higher, ranging between 10 and 20.

9 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 [18], 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. 8, 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 [43], 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 [45] 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. [46] present an automated verifier for the RSL family of logics [13, 40, 47] for reasoning about weak-memory concur-

rency. 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 [11–13, 40, 47–51] 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. Diaframe [52] introduces a proof search strategy for Iris [53], achieving foundational proofs and a high degree of automation. This strategy applies rules based on the syntactic shape of the verification goal. To improve completeness, users can provide hints that specify how certain goals are split into subgoals. In contrast to TaDA and our work, Diaframe does not support abstract atomicity.

Starling [22] 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 [34] 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 [17] 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 [23] 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.

10 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 combination 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-specialized 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 [37, 54].

Acknowledgements We thank the anonymous referees of this paper and the earlier conference paper 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 TaDA and for feedback on Voila.

Funding Open access funding provided by Swiss Federal Institute of Technology Zurich.

Data availability An artifact with the executable Voila tool, including source code and all our test and evaluation programs, is openly available [26]. The Voila source repository is also available [27]. The measurements taken for the evaluation are available from the corresponding author upon request.

Appendix A: Full TaDA proof

Figure 8 shows the full TaDA proof of the TaDA proof outline from Fig. 1. The purpose of the figure is to illustrate the complexity of full TaDA proofs. We do not expect readers to be able to understand the proof. All parts of the proof that are present in the TaDA proof outline are colored in blue. Everything else is inferred by Voila.

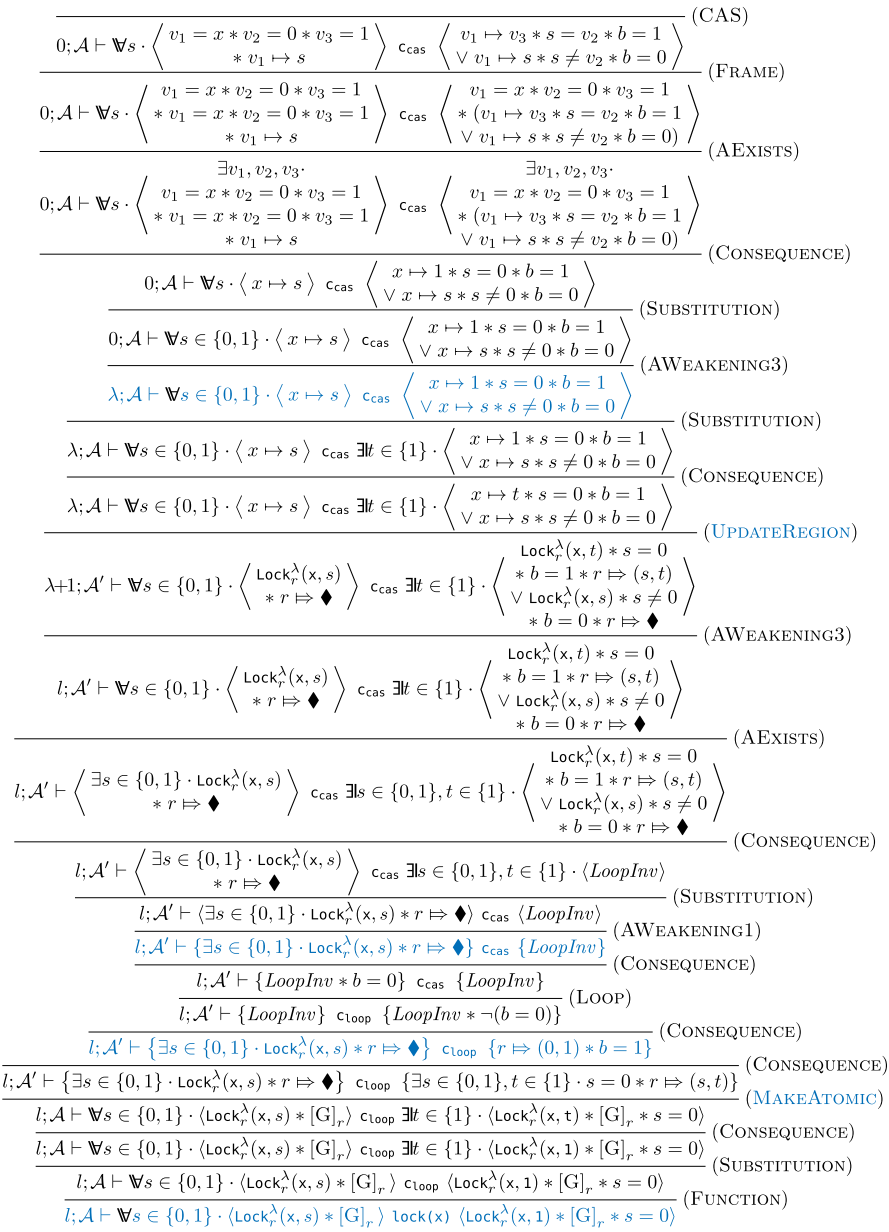

Fig. 8 Simplified version of the full TaDA proof for the TaDA proof outline from Fig. 1. The parts that are present in the TaDA proof outline are colored blue. The statements $\texttt{c}_{\texttt{loop}}$ and $\texttt{c}_{\texttt{cas}}$ are the loop and CAS statement, respectively. The loop invariant *LoopInv* is $\exists s \in \{0,1\} \cdot \mathsf{Lock}^{\lambda}(x,s) * (r \Mapsto (0,1) * b = 1 \vee r \Mapsto \blacklozenge * b = 0)$. The atomicity context $\mathcal{A}'$ is $r : s \in \{0,1\} \rightsquigarrow 1, \mathcal{A}$. For simplicity, redundant quantifiers are omitted and local variables are not put into the private part of atomic assertions. Furthermore, the proof uses a variation of the MakeAtomic rule that can be derived in TaDA

References

1. O'Hearn PW, Reynolds JC, Yang H (2001) Local reasoning about programs that alter data structures. In: CSL, vol 2142. Lecture notes in computer science. Springer, New York, pp 1–19
2. Reynolds JC (2002) Separation logic: a logic for shared mutable data structures. In: LICS. IEEE Computer Society, New York, pp 55–74
3. O'Hearn PW (2004) Resources, concurrency and local reasoning. In: CONCUR, vol 3170. Lecture notes in computer science. Springer, New York, pp 49–67
4. Brookes SD (2004) A semantics for concurrent separation logic. In: CONCUR, vol 3170. Lecture notes in computer science. Springer, New York, pp 16–34
5. Owicki SS, Gries D (1976) An axiomatic proof technique for parallel programs I. Acta Inf 6:319–340
6. Jones CB (1983) Specification and design of (parallel) programs. In: IFIP congress, pp 321–332
7. Dinsdale-Young T, Dodds M, Gardner P, Parkinson MJ, Vafeiadis V (2010) Concurrent abstract predicates. In: ECOOP, vol 6183. Lecture notes in computer science. Springer, New York, pp 504–528
8. Svendsen K, Birkedal L (2014) Impredicative concurrent abstract predicates. In: Shao Z (ed) European symposium on programming (ESOP), vol 8410. Lecture notes in computer science. Springer, New York, pp 149–168
9. Turon A, Dreyer D, Birkedal L (2013) 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). ACM, New York, pp 377–390
10. Raad A, Villard J, Gardner P (2015) CoLoSL: concurrent local subjective logic. In: Vitek J (ed) ESOP, vol 9032. Lecture notes in computer science. Springer, New York, pp 710–735
11. Sergey I, Nanevski A, Banerjee A (2015) Mechanized verification of fine-grained concurrent programs. In: PLDI. ACM, New York, pp 77–87
12. Turon A, Vafeiadis V, Dreyer D (2014) GPS: navigating weak memory with ghosts, protocols, and separation. In: OOPSLA. ACM, New York, pp 691–707
13. Vafeiadis V, Narayan C (2013) Relaxed separation logic: a program logic for C11 concurrency. In: OOPSLA. ACM, New York, pp 867–884
14. da Rocha Pinto P, Dinsdale-Young T, Gardner P (2014) TaDA: a logic for time and data abstraction. In: ECOOP, vol 8586. Lecture notes in computer science. Springer, New York, pp 207–231
15. Brookes S, O'Hearn PW (2016) Concurrent separation logic. SIGLOG News 3(3):47–65
16. Berdine J, Calcagno C, O'Hearn PW (2005) Smallfoot: modular automatic assertion checking with separation logic. In: FMCO, vol 4111. Lecture notes in computer science. Springer, New York, pp 115–137
17. Piskac R, Wies T, Zufferey D (2014) GRASShopper—complete heap verification with mixed specifications. In: TACAS, vol 8413. Lecture notes in computer science. Springer, New York, pp 124–139
18. Dinsdale-Young T, da Rocha Pinto P, Andersen KJ, Birkedal L (2017) Caper—automatic verification for fine-grained concurrency. In: ESOP, vol 10201. Lecture notes in computer science. Springer, New York, pp 420–447
19. Owicki SS (1975) Axiomatic proof techniques for parallel programs. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York
20. Apt KR, de Boer FS, Olderog E (2009) Verification of sequential and concurrent programs. Texts in computer science. Springer, New York
21. Pierce BC, Azevedo de Amorim A, Casinghino C, Gaboardi M, Greenberg M, Hriţcu C, Sjöberg V, Tolmach A, Yorgey B (2018) Programming language foundations, vol 2. Software foundations series. Electronic Textbook, Pennsylvania
22. Windsor M, Dodds M, Simner B, Parkinson MJ (2017) Starling: lightweight concurrency verification with views. In: CAV, vol 10426. Lecture notes in computer science. Springer, New York, pp 544–569
23. Jacobs B, Smans J, Philippaerts P, Vogels F, Penninckx W, Piessens F (2011) VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: NASA formal methods, vol 6617. Lecture notes in computer science. Springer, New York, pp 41–55
24. da Rocha Pinto P (2016) Reasoning with time and data abstractions. PhD thesis, Imperial College London, UK
25. Müller P, Schwerhoff M, Summers AJ (2016) Viper: a verification infrastructure for permission-based reasoning. In: VMCAI, vol 9583. Lecture notes in computer science. Springer, New York, pp 41–62
26. Wolf FA, Schwerhoff M, Müller P Concise outlines for a complex logic: a proof outline checker for TaDA. https://doi.org/10.5281/zenodo.5137791
27. Wolf FA, Schwerhoff M, Müller P The Voila source repository. https://github.com/viperproject/voila Accessed 2021-07-26
28. Wolf FA, Schwerhoff M, Müller P (2020) Concise outlines for a complex logic: a proof outline checker for TaDA (full paper). CoRR arXiv:2010.07080

29. Wolf FA, Schwerhoff M, Müller P (2021) Concise outlines for a complex logic: a proof outline checker for TaDa. In: FM, vol 13047. Lecture notes in computer science. Springer, New York, pp 407–426
30. Parkinson MJ, Bierman GM (2005) Separation logic and abstraction. In: POPL. ACM, New York, pp 247–258
31. Herlihy M, Wing JM (1990) Linearizability: a correctness condition for concurrent objects. ACM Trans Program Lang Syst 12(3):463–492
32. Mooij AJ, Wesselink W (2005) Incremental verification of Owicki/Gries proof outlines using PVS. In: Lau K, Banach R (eds) International conference on formal engineering methods (ICFEM), vol 3785. Lecture notes in computer science. Springer, New York, pp 390–404
33. Leino KRM (2010) Dafny: An automatic program verifier for functional correctness. In: Clarke EM, Voronkov A (eds) Logic for programming, artificial intelligence, and reasoning (LPAR), vol 6355. Lecture notes in computer science. Springer, New York, pp 348–370
34. Dinsdale-Young T, Birkedal L, Gardner P, Parkinson MJ, Yang H (2013) Views: compositional reasoning for concurrent programs. In: POPL. ACM, New York, pp 287–300
35. Smans J, Jacobs B, Piessens F (2009) Implicit dynamic frames: combining dynamic frames and separation logic. ECOOP, vol 5653. Lecture notes in computer science. Springer, New York, pp 148–172
36. Parkinson MJ, Summers AJ (2012) The relationship between separation logic and implicit dynamic frames. Log Methods Comput Sci 8(3:01):1–54
37. da Rocha Pinto P, Dinsdale-Young T, Gardner P, Sutherland J (2016) Modular termination verification for non-blocking concurrency. In: ESOP, vol 9632. Lecture notes in computer science. Springer, New York, pp 176–201
38. Boyland J (2003) Checking interference with fractional permissions. In: SAS, vol 2694. Lecture notes in computer science. Springer, New York, pp 55–72
39. Treiber RK (1986) Systems programming: coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center
40. Doko M, Vafeiadis V (2017) Tackling real-life relaxed concurrency with FSL++. In: ESOP, vol 10201. Lecture notes in computer science. Springer, New York, pp 448–475
41. Kaiser J, Dang H, Dreyer D, Lahav O, Vafeiadis V (2017) Strong logic for weak memory: reasoning about release-acquire consistency in Iris. In: ECOOP. LIPIcs, vol 74. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, Wadern, pp 17:1–17:29
42. 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 (2009) seL4: formal verification of an OS kernel. In: SOSP. ACM, New York, pp 207–220
43. Calcagno C, Parkinson MJ, Vafeiadis V (2007) Modular safety checking for fine-grained concurrency. In: SAS, vol 4634. Lecture notes in computer science. Springer, New York, pp 233–248
44. Vafeiadis V (2010) Automatically proving linearizability. In: CAV, vol 6174. Lecture notes in computer science. Springer, New York, pp 450–464
45. Oortwijn W, Blom S, Gurov D, Huisman M, Zaharieva-Stojanovski M (2017) An abstraction technique for describing concurrent program behaviour. In: VSTTE, vol 10712. Lecture notes in computer science. Springer, New York, pp 191–209
46. Summers AJ, Müller P (2018) Automating deductive verification for weak-memory programs. In: TACAS (1), vol 10805. Lecture notes in computer science. Springer, New York, pp 190–209
47. Doko M, Vafeiadis V (2016) A program logic for C11 memory fences. In: VMCAI, vol 9583. Lecture notes in computer science. Springer, New York, pp 413–430
48. Nanevski A., Ley-Wild R, Sergey I, Delbianco GA (2014) Communicating state transition systems for fine-grained concurrent resources. In: ESOP, vol 8410. Lecture notes in computer science. Springer, New York, pp 290–310
49. Frumin D, Krebbers R, Birkedal L (2018) ReLoC: a mechanised relational logic for fine-grained concurrency. In: LICS. ACM, New York, pp 442–451
50. Krebbers R, Jourdan J, Jung R, Tassarotti J, Kaiser J, Timany A, Charguéraud A, Dreyer D (2018) Mosel: a general, extensible modal framework for interactive proofs in separation logic. PACMPL 2(ICFP) 77:1–77:30
51. Jung R, Krebbers R, Jourdan J, Bizjak A, Birkedal L, Dreyer D (2018) Iris from the ground up: a modular foundation for higher-order concurrent separation logic. J Funct Program 28:20
52. Mulder I, Krebbers R, Geuvers H (2022) Diaframe: automated verification of fine-grained concurrent programs in Iris. In: PLDI. ACM, New York, pp 809–824
53. Jung R, Swasey D, Sieczkowski F, Svendsen K, Turon A, Birkedal L, Dreyer D (2015) Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In: POPL. ACM, New York, pp 637–650
54. D'Osualdo E, Farzan A, Gardner P, Sutherland J (2019) TaDA live: compositional reasoning for termination of fine-grained concurrent programs. CoRR arXiv:1901.05750