Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA (Full Paper)

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 [27,35] and data-race free concurrent programs [26,5]. 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 [30], and rely-guarantee [16]. Examples include CAP [8], iCAP [43], CaReSL [45], CoLoSL [34], FCSL [39], GPS [46], RSL [48], and TaDA [37] (see Brookes et al. [4] 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 [48] and FCSL [39]. 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 [33] for traditional separation logics, and Caper [9] 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* [29,1], 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 [49] and VeriFast [15] 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 [37], 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 [37,36] (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 [24], 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 [51], and the Voila source repository is located at [50].

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 appendix contains 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 [37], 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 [16]). A shared region such as $\mathtt{Lock}_r(\mathtt{x}, s)$ is an abstraction over the region's content, analogous to abstract predicates [32] 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 $\mathtt{Lock}_r$ 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

1 $I(\mathbf{Lock}_r(x,0)) \triangleq x \mapsto 0$
2 $I(\mathbf{Lock}_r(x,1)) \triangleq x \mapsto 1$

---

3 $\mathrm{G} : 0 \rightsquigarrow 1$
4 $\mathrm{G} : 1 \rightsquigarrow 0$

---

5 $\mathrm{G} \bullet \mathrm{G}$ is undefined

---

6 $\forall s \in \{0,1\}.$
7 $\langle \mathbf{Lock}_r(\mathtt{x}, s) * [\mathrm{G}]_r \rangle$
8 MakeAtomic: $r : s \in \{0,1\} \rightsquigarrow 1 \vdash$
9 $\{\exists s \in \{0,1\}.\, \mathbf{Lock}_r(\mathtt{x}, s) * r \Mapsto \blacklozenge\}$
10 `do` {
11 $\{\exists s \in \{0,1\}.\, \mathbf{Lock}_r(\mathtt{x}, s) * r \Mapsto \blacklozenge\}$
12 UpdateRegion: $\forall s \in \{0,1\}.$
13 $\langle \mathtt{x} \mapsto s \rangle$
14 $\mathtt{b} := \mathtt{CAS}(\mathtt{x}, 0, 1);$
15 $\langle (\mathtt{x} \mapsto 1 * s = 0 * \mathtt{b} = 1) \vee$
16 $(\mathtt{x} \mapsto s * s \neq 0 * \mathtt{b} = 0) \rangle$
17 $\{\exists s \in \{0,1\}.\, \mathbf{Lock}_r(\mathtt{x}, s) *$
18 $((r \Mapsto (0,1) * \mathtt{b} = 1) \vee$
19 $(r \Mapsto \blacklozenge * \mathtt{b} = 0))\}$
20 } `while` $(\mathtt{b} = 0);$
21 $\{r \Mapsto (0,1) * \mathtt{b} = 1 * [\mathrm{G}]_r\}$
22 $\langle \mathbf{Lock}_r(\mathtt{x}, 1) * [\mathrm{G}]_r * s = 0 \rangle$

Fig. 1: TaDA spinlock example with shared region `Lock`; adapted with only minor changes from TaDA [37]. 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. [37], verifiable by Voila and shown in App. D) 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 App. D.

threads to compete for the lock; in this example, the usual lock semantics is established by an enclosing region (`CAPLock` [37]; see App. D).

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)

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

Atomic triples (angle brackets) express that their statement is linearizable [14]. 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 ($\mathtt{Lock}_r(\mathtt{x}, \mathtt{1})$). In general, callers must assume that a region's abstract state may have been changed by the environment after the linearization point

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

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

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* $r \Mapsto (0, 1)$, 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 [29], or tools that automatically check the validity of a proof outline [15,23,49]; 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 [22] 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 [24], 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 App. C, 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 [9]); additional ones can be encoded, see App. H. 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 $\mathbb{V}s \in \{0, 1\}$.

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 [37,36] 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* [7,36] – 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 [36] 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 App. F 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 [24]. 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 [40,31] 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 [32] 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

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

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

foreach g(p̄': t̄') ∈ G:
  predicate R_g(r: Ref, p̄':⟦t̄'⟧)
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).

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 $r \Mapsto \blacklozenge$.

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.

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

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 App. G.3.

7 Evaluation

We evaluated Voila on nine benchmark examples from Caper's test suite, with the Treiber's stack [44] 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 App. B). 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 [11,19,20,48] 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 [9], from which we drew inspiration, e.g. for how to specify region transition systems. Caper supports an improved version of CAP [8], 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 [6], which can prove memory safety, but not functional correctness, and CAVE [47], which can prove linearizability, but cannot reason about non-linearizable code (which TaDA and Voila can). VerCors [28] 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. [42] present an automated verifier for the RSL family of logics [48,10,11] 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 [48,25,46,39,10,11,13,21,17] 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 [49] 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 [7] 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 [33] 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 [15] 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 [38,12].

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. Boyland, J.: Checking interference with fractional permissions. In: SAS. Lecture Notes in Computer Science, vol. 2694, pp. 55–72. Springer (2003)
4. Brookes, S., O'Hearn, P.W.: Concurrent separation logic. SIGLOG News **3**(3), 47–65 (2016)
5. Brookes, S.D.: A semantics for concurrent separation logic. In: CONCUR. Lecture Notes in Computer Science, vol. 3170, pp. 16–34. Springer (2004)
6. 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)
7. 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)
8. 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)
9. 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)
10. 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)
11. 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)
12. 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)
13. Frumin, D., Krebbers, R., Birkedal, L.: Reloc: A mechanised relational logic for fine-grained concurrency. In: LICS. pp. 442–451. ACM (2018)
14. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
15. 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)
16. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress. pp. 321–332 (1983)
17. 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)
18. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: POPL. pp. 637–650. ACM (2015)
19. 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)

20. 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)
21. 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)
22. 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)
23. 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)
24. 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)
25. 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)
26. O'Hearn, P.W.: Resources, concurrency and local reasoning. In: CONCUR. Lecture Notes in Computer Science, vol. 3170, pp. 49–67. Springer (2004)
27. 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)
28. 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)
29. Owicki, S.S.: Axiomatic Proof Techniques for Parallel Programs. Outstanding Dissertations in the Computer Sciences, Garland Publishing, New York (1975)
30. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inf. **6**, 319–340 (1976)
31. 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)
32. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: POPL. pp. 247–258. ACM (2005)
33. 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)
34. 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)
35. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE Computer Society (2002)
36. da Rocha Pinto, P.: Reasoning with time and data abstractions. Ph.D. thesis, Imperial College London, UK (2016)
37. 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)

38. 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)
39. Sergey, I., Nanevski, A., Banerjee, A.: Mechanized verification of fine-grained concurrent programs. In: PLDI. pp. 77–87. ACM (2015)
40. 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)
41. Summers, A.J., Drossopoulou, S.: A formal semantics for isorecursive and equirecursive state abstractions. In: European Conference on Object-Oriented Programming. pp. 129–153. Springer (2013)
42. 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)
43. 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)
44. Treiber, R.K.: Systems programming: Coping with parallelism. Tech. Rep. RJ 5118, IBM Almaden Research Center (1986)
45. 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)
46. Turon, A., Vafeiadis, V., Dreyer, D.: GPS: navigating weak memory with ghosts, protocols, and separation. In: OOPSLA. pp. 691–707. ACM (2014)
47. Vafeiadis, V.: Automatically proving linearizability. In: CAV. Lecture Notes in Computer Science, vol. 6174, pp. 450–464. Springer (2010)
48. Vafeiadis, V., Narayan, C.: Relaxed separation logic: a program logic for C11 concurrency. In: OOPSLA. pp. 867–884. ACM (2013)
49. 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)
50. Wolf, F.A., Schwerhoff, M., Müller, P.: The Voila source repository, `https://github.com/viperproject/voila`
51. 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

A TaDA Key Proof Rules

$$\textsc{OpenRegion}\quad \frac{\Gamma;\lambda;\mathcal{A}\vdash \mathbb{V}x\in X.\langle P_p \mid I(\mathbf{t}_a^\lambda(\vec{z},x))*P(x)\rangle\ \mathbb{C}\quad \exists\!\!\mathbb{I} y\in Y.\langle Q_p(x,y)\mid I(\mathbf{t}_a^\lambda(\vec{z},x))*Q(x,y)\rangle}{\Gamma;\lambda+1;\mathcal{A}\vdash \mathbb{V}x\in X.\langle P_p \mid \mathbf{t}_a^\lambda(\vec{z},x)*P(x)\rangle\ \mathbb{C}\quad \exists\!\!\mathbb{I} y\in Y.\langle Q_p(x,y)\mid \mathbf{t}_a^\lambda(\vec{z},x)*Q(x,y)\rangle}$$

$$\textsc{UseAtomic}\quad \frac{a\notin\mathcal{A}\qquad \forall x\in X.\,(x,f(x))\in\mathcal{T}_{\mathbf{t}}(\mathrm{G})^*\qquad \Gamma;\lambda;\mathcal{A}\vdash \mathbb{V}x\in X.\langle P_p \mid I(\mathbf{t}_a^\lambda(\vec{z},x))*P(x)*[\mathrm{G}]_a\rangle\ \mathbb{C}\quad \exists\!\!\mathbb{I} y\in Y.\langle Q_p(x,y)\mid I(\mathbf{t}_a^\lambda(\vec{z},f(x)))*Q(x,y)\rangle}{\Gamma;\lambda+1;\mathcal{A}\vdash \mathbb{V}x\in X.\langle P_p \mid \mathbf{t}_a^\lambda(\vec{z},x)*P(x)*[\mathrm{G}]_a\rangle\ \mathbb{C}\quad \exists\!\!\mathbb{I} y\in Y.\langle Q_p(x,y)\mid \mathbf{t}_a^\lambda(\vec{z},f(x))*Q(x,y)\rangle}$$

$$\textsc{UpdateRegion}\quad \frac{\Gamma;\lambda;\mathcal{A}\vdash \mathbb{V}x\in X.\left\langle P_p \,\middle|\, I(\mathbf{t}_a^\lambda(\vec{z},x))*P(x)\right\rangle\ \mathbb{C}\quad \exists\!\!\mathbb{I}(y,w)\in Y\times W.\left\langle Q_p(x,y,w)\,\middle|\,\begin{array}{l} I(\mathbf{t}_a^\lambda(\vec{z},w))*Q_1(x,y,w)\\ \vee\, I(\mathbf{t}_a^\lambda(\vec{z},x))*Q_2(x,y)\end{array}\right\rangle}{\begin{array}{c}\mathbb{V}x\in X.\langle P_p \mid \mathbf{t}_a^\lambda(\vec{z},x)*P(x)*a\Mapsto\blacklozenge\rangle\\ \Gamma;\lambda+1;a:x\in X\rightsquigarrow W,\mathcal{A}\vdash \begin{array}{c}\mathbb{C}\\ \exists\!\!\mathbb{I}(y,w)\in Y\times W.\left\langle Q_p(x,y,w)\,\middle|\,\begin{array}{l}\mathbf{t}_a^\lambda(\vec{z},w)*Q_1(x,y,w)*a\Mapsto(x,w)\\ \vee\,\mathbf{t}_a^\lambda(\vec{z},x)*Q_2(x,y)*a\Mapsto\blacklozenge\end{array}\right\rangle\end{array}\end{array}}$$

$$\textsc{MakeAtomic}\quad \frac{a\notin\mathcal{A}\qquad \{(x,y)\mid x\in X,y\in Y\}\subseteq\mathcal{T}_{\mathbf{t}}(\mathrm{G})^*\qquad \Gamma;\lambda';a:x\in X\rightsquigarrow Y,\mathcal{A}\vdash\{P_p*\exists x\in X.\,\mathbf{t}_a^\lambda(\vec{z},x)*a\Mapsto\blacklozenge\}\ \mathbb{C}\ \{\exists x\in X,y\in Y.\,Q_p(x,y)*a\Mapsto(x,y)\}}{\Gamma;\lambda';\mathcal{A}\vdash \mathbb{V}x\in X.\langle P_p \mid \mathbf{t}_a^\lambda(\vec{z},x)*[\mathrm{G}]_a\rangle\ \mathbb{C}\quad \exists\!\!\mathbb{I} y\in Y.\langle Q_p(x,y)\mid \mathbf{t}_a^\lambda(\vec{z},y)*[\mathrm{G}]_a\rangle}$$

Fig. 6: Fig. 6 shows TaDA's key proof rules as they are presented in da Rocha Pinto's thesis [36], including public and private assertions, and region levels. Levels are omitted from the discussion in this paper, but supported by Voila. The combination of public and private assertions in a rule triple is currently not supported by Voila.

B Supported TaDA Ingredients

Fig. 7 provides an overview of Voila's features, w.r.t. TaDA ingredients and Caper guard algebras [9]. The left column lists TaDA features and to which extent their incur annotation overhead. *None* means that the ingredient does not surface at all in a Voila program. *Once* means that there is a one-time annotation per Voila program, typically in the form of a background declaration such as a region. In contrast, *proc* means that the feature requires a one-time annotation per Voila procedure, typically as part of a procedure specification. Next, *low* means that the feature may result in more than one annotation per procedure: for regions, these are new-region statements (one per newly created region instance), in addition to region declarations. Tracking resources, on the other hand, typically appear in invariants of loops that repeat until an update succeeded. Finally, View shifts incur a *medium* annotation overhead: most standard view shifts are automated by Voila and do not require annotations, but for complex, manually encoded examples, additional annotations may be required. See also App. H.

Most of Caper's guard algebras are supported by Voila, and as such, do not incur any additional overhead (the guards themselves must still be mentioned, e.g. in specifications). Only counting and sum guards are not directly supported by Voila; they can be encoded, which will require additional annotations. See also App. H for an example of a manually encoded guard algebra.

Ingredient	Annotations
Regions	low
Transition systems	once
Triple kinds	proc
Interference contexts	proc
Atomicity contexts	none
Levels	proc
Tracking resources	low
Private vs. public	–
View Shifts	medium
Stability	none
Framing	none

Guard Algebra	Support
Trivial	built-in
All-or-nothing	built-in
Counting	encodable
Indexed	built-in
Product	built-in
Permissions	built-in
Sum	encodable

Fig. 7: Supported TaDA ingredients, with a classification of the incurred annotation overhead, and Caper guard algebras [9], with a classification of their support. TaDA's combination of public and private assertions in a rule triple is currently not supported by Voila.

C Voila Grammar

This section gives an overview of Voila's grammar, and shows that Voila strongly resembles TaDA, but requires fewer technical details in its annotation language.

$$
\begin{aligned}
t &::= \texttt{id} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{frac} \mid S \\
e &::= x \mid \texttt{?}x \mid l \mid e \texttt{ \&\& } e \mid e \texttt{ || } e \mid \texttt{!}e \mid e \Rightarrow e \mid e \ op \ e \\
a &::= e \mid x.f \mapsto e \mid a \texttt{ \&\& } a \mid e \Rightarrow a \mid R\texttt{(}r, \overline{e}\texttt{)} \mid G(\overline{e})\texttt{@}r \mid r \mapsto \blacklozenge \mid r \mapsto \texttt{(}e, e\texttt{)}
\end{aligned}
$$

Fig. 8: Voila's core syntax for types t, expressions e, and assertions a). Types t include the type of region identifiers `id`, fractions `frac`, and struct types S. Expressions e include variables x, literals l, fields f, and the usual expression operators, e.g. relational ones. They also include variable binders `?`x, which are allowed in only two places: the right-hand side of points-to assertions and the last parameter of a region instance, binding the region's abstract state. Assertions a include, besides the usual separation logic assertions, region instances $R(r, \overline{e})$, where R denotes a region name, r a region identifier, and $\overline{e}$ the region's abstract state; the last argument may be omitted when the region state is unspecified. As usual, overlines denote lists. Moreover, assertions include guards $G(\overline{e})@r$, where G denotes a guard name and $\overline{e}$ the guard arguments (guards without arguments are written as $G@r$), and TaDA's two tracking resources. For brevity, we omitted levels, collection data types (i.e. sets, sequence, maps, and tuples), and more complex guards (but see also App. B and App. H).

```
ns ::= x := e                          as ::= x.f := e
     | x̄ := P(ē)                            | y := x.f
     | if (e) {s} else {s}                  | x̄ := P(ē)
     | while (e) invariant a {s}            | use_atomic using R(r, ē) with G(ē)@r {as}
     | s; s                                 | make_atomic using R(r, ē) with G(ē)@r {s}
                                            | open_region using R(r, ē) {as}
 s ::= t x | ns | as                        | update_region using R(r, ē) {as}
```

Fig. 9: Voila's core syntax for statements s, atomic statements as, and non-atomic statements ns. Statements s comprise variable declarations as well as atomic and non-atomic statements; the categorization of the latter follows TaDA. Atomic statements as include field reads and writes, invocations of abstract-atomic procedures, and key rule statements. Following TaDA, rule statements other than `make_atomic` may only nest atomic statements. Non-atomic statements ns are local variable assignments, invocations of non-atomic procedures, and compound statements. For brevity, statements for creating struct and region instances have been omitted, as have ghost statements useful for encoding, e.g. complex guard algebras (see App. H).

```
struct S {t̄ f̄}     region R(id r, t̄ x̄)          abstract_atomic procedure P(t̄ x̄)
                      interpretation {a}              returns (t̄ ȳ)
                      state {e}                     interference ?x in e
                      guards {mod G(t̄ x̄)}           requires a
                      actions {G(ē): e ⇝ e}         ensures a
                                                  {s̄}
```

Fig. 10: Voila's core syntax for struct, region, and procedure declarations. Structs declare fields and induce homonymous types. Region declarations include name R, identifier r and further formal arguments $\overline{t\ x}$. A region's interpretation and state are an assertion and expression, respectively. Each region may declare guards $G(\overline{t\ x})$, with formal arguments $\overline{x}$ and modifier `unique` or `duplicable`, and actions that describe possible state changes. Abstract-atomic procedure declarations include an interference clause that corresponds to TaDA's interference context. More complex guard and action definitions are omitted for brevity, as are non-atomic and lemma procedures (but see also App. D and App. H).

D Full Lock and CAPLock Example

This section complements our running example by showing TaDA outline and Voila code for (1) region `Lock` and procedure `lock`, but with the previously omitted levels, and (2) region `CAPLock` and procedure `acquire`, which build on the former and provide the expected lock semantics.

$$I(\mathbf{Lock}^0_r(x,0)) \triangleq x \mapsto 0$$
$$I(\mathbf{Lock}^0_r(x,1)) \triangleq x \mapsto 1$$

$$\mathrm{G} : 0 \rightsquigarrow 1$$
$$\mathrm{G} : 1 \rightsquigarrow 0$$

$$\mathrm{G} \bullet \mathrm{G} \text{ is undefined}$$

$$\mathbb{\forall} s \in \{0,1\}.$$
$$\langle \mathbf{Lock}^0_r(\mathtt{x}, s) * [\mathrm{G}]_r \rangle$$
$$r : s \in \{0,1\} \rightsquigarrow 1 \vdash$$
$$\{\exists s \in \{0,1\}.\, \mathbf{Lock}^0_r(\mathtt{x}, s) * r \mapsto \blacklozenge\}$$
$$\mathtt{do}\ \{$$
$$\{\exists s \in \{0,1\}.\, \mathbf{Lock}^0_r(\mathtt{x}, s) * r \mapsto \blacklozenge\}$$
$$\mathbb{\forall} s \in \{0,1\}.$$
$$\langle \mathtt{x} \mapsto s \rangle$$
$$\mathtt{b} := \mathtt{CAS}(\mathtt{x}, 0, 1);$$
$$\left\langle \begin{array}{l} (\mathtt{x} \mapsto 1 * s = 0 * \mathtt{b} = 1) \vee \\ (\mathtt{x} \mapsto s * s \neq 0 * \mathtt{b} = 0) \end{array} \right\rangle$$
$$\left\{ \begin{array}{l} \exists s \in \{0,1\}.\, \mathbf{Lock}^0_r(\mathtt{x}, s) * \\ ((r \mapsto (0,1) * \mathtt{b} = 1) \vee \\ (r \mapsto \blacklozenge * \mathtt{b} = 0)) \end{array} \right\}$$
$$\}\ \mathtt{while}\ (\mathtt{b} = 0);$$
$$\{r \mapsto (0,1) * \mathtt{b} = 1 * [\mathrm{G}]_r\}$$
$$\langle \mathbf{Lock}^0_r(\mathtt{x}, 1) * [\mathrm{G}]_r * s = 0 \rangle$$

MakeAtomic

UpdateRegion

$$I(\mathbf{CAPLock}^\lambda_a(r,x,v)) \triangleq \mathbf{Lock}^0_r(x,0) * [\mathrm{G}]_r * [\mathrm{U}]_a$$
$$I(\mathbf{CAPLock}^\lambda_a(r,x,v)) \triangleq \mathbf{Lock}^0_r(x,1) * [\mathrm{G}]_r$$

$$\mathbf{0} : 0 \rightsquigarrow 1$$
$$\mathrm{U} : 1 \rightsquigarrow 0$$

$$\mathrm{U} \bullet \mathrm{U} \text{ is undefined}$$

$$\left\{\exists v \in \{0,1\}.\, \mathbf{CAPLock}^\lambda_a(r, \mathtt{x}, v)\right\}$$
$$\mathbb{\forall} v \in \{0,1\}.$$
$$\left\langle \begin{array}{l} (\mathbf{Lock}^0_r(\mathtt{x}, 0) * [\mathrm{G}]_r * [\mathrm{U}]_a * v = 0) \vee \\ (\mathbf{Lock}^0_r(\mathtt{x}, 1) * [\mathrm{G}]_r * v = 1) \end{array} \right\rangle$$
$$\mathbb{\forall} s \in \{0,1\}.$$
$$\langle \mathbf{Lock}^0_r(\mathtt{x}, s) * [\mathrm{G}]_r \rangle$$
$$\mathtt{lock}(\mathtt{x})$$
$$\langle \mathbf{Lock}^0_r(\mathtt{x}, 1) * [\mathrm{G}]_r * s = 0 \rangle$$
$$\langle \mathbf{Lock}^0_r(\mathtt{x}, 1) * [\mathrm{G}]_r * [\mathrm{U}]_a \rangle$$
$$\left\{\mathbf{CAPLock}^\lambda_a(r, \mathtt{x}, 1) * [\mathrm{U}]_a\right\}$$

UseAtomic

Frame: $v = 0 \rightarrow [\mathrm{U}]_a$

Fig. 11: TaDA declarations and proof outlines adapted from [36]: the left column repeats our running example (region `Lock`, proof outline for procedure `lock`), but with levels included. The right column shows the `CAPLock` region and a proof outline for procedure `acquire`, which build on `Lock` and `lock`, respectively. The `CAPLock` abstraction provides the expected lock semantics, via its guards and actions: the vacuous zero guard **`0`** allows arbitrarily many clients to compete for the lock (i.e. call `acquire`), but only the holder of the unique U guard can release the lock again. Proof outlines of the latter procedures (`release`/`unlock` for `CAPLock`/`Lock`) are straightforward and have been omitted.

The TaDA triple proved by the proof outline in the left (body of procedure `lock`) and right (body of procedure `acquire`) column, respectively, are the following:

$$\mathcal{A} \vdash \mathbb{\forall} s \in \{0,1\} \cdot \langle \mathtt{Lock}^0_r(\mathtt{x}, s) * [\mathrm{G}]_r \rangle\ \mathtt{lock(x)}\ \langle \mathtt{Lock}^0_r(\mathtt{x}, \mathtt{1}) * [\mathrm{G}]_r * s = 0 \rangle$$
$$\lambda; \mathcal{A} \vdash \{\exists v \in \{0,1\} \cdot \mathtt{CAPLock}^\lambda_a(r, \mathtt{x}, v)\}\ \mathtt{acquire(x)}\ \{\mathtt{CAPLock}^\lambda_a(r, \mathtt{x}, \mathtt{1}) * [\mathrm{U}]_a\}$$

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

procedure acquire(id a, int lvl, id r, cell x)
  requires CAPLock(a, lvl, r, x) && Z@a;
  ensures  CAPLock(a, lvl, r, x, 1) && U@a;
{
  use_atomic using CAPLock(a,lvl, r, x) with Z@a {
    lock(r, 0, x);
  }
}

// Repetition of our running example, but with previously omitted levels

struct cell {
  int val;
}

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

abstract_atomic procedure lock(id r, int lvl, cell x)
  interference ?s in Set(0, 1);
  requires Lock(r, lvl, x, s) && G@r;
  ensures Lock(r, lvl, x, 1) && G@r && s == 0;
```

Fig. 12: The Voila proof outline of TaDA's CAPLock [37], building on our lock running example Fig. 3. Voila does not yet support TaDA's zero guard; instead, we use a duplicable guard Z. Following Fig. 11, `CAPLock` uses `Lock` with a fixed level of 0, but Voila also verifiers a more general version, where `Lock`'s level is any level smaller than `CAPLock`'s. Also following the TaDA source, `Lock`'s identifier `r` is exposed as an argument to `CAPLock`. An alternative would be to existentially quantify it; in Voila, this can be modeled via a ghost field of `CAPLock`.

E Extended Discussion of our Normal Form

Recall from Sec. 5 that we impose a normal form on the rule triples of our proof candidate, with four main restrictions: triples are exclusively atomic or non-atomic; all triple preconditions, as well as the postconditions of non-atomic triples, are stable; in atomic triples, the state of every region in the precondition is bound by exactly one interference quantifier; and triples must hold for a range of atomicity contexts. The normal form is required to hold for the premises and conclusions of all syntax-driven and key rules, which allows our heuristics to exploit the restrictions when inserting applications of bridge rules. Bridge rules themselves may violate the normal form, which increases completeness.

Next, we provide additional details on the last normal form restriction: triples must hold for a heuristically determined *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: most importantly, by enabling modular procedure specifications, which, as confirmed by the TaDA authors, was not possible in the original logic.

TaDA proofs require a suitable instantiation of the atomicity context $\mathcal{A}$, i.e., the set of pending region updates. Choosing a set that is too small provides weak stability guarantees and thus, leads to unnecessary weakening of assertions, whereas a set that is too large prevents certain applications of the MakeAtomic rule. In both cases, the proof may fail even for correct programs. Moreover, for procedure specifications, it is virtually impossible to chose a single atomicity $\mathcal{A}$ that allows all possible clients to call the procedure, since each client would have to establish *exactly* $\mathcal{A}$,

To overcome these problems, Voila proves triples for *all* atomicity contexts within certain bounds. These bounds are inferred by proof state already present in the proof candidate, by partitioning the set of currently held region instances into two sets: the first contains all regions that the triple's statement may update; this set corresponds to the lower bound, and is manipulated according to MakeAtomic. The second set contains all regions that the code to verify cannot update anyway; it corresponds to the upper bound, and is determined based on level information.

F Extended Discussion of our Heuristics

Recall from Sec. 5 that our heuristics infer bridge rule applications locally, by inspecting only adjacent rule applications that are to be composed, and their proof state. 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. The first three heuristics have been described briefly in Sec. 5; here, we provide additional details and illustrate some of the heuristics on our running example. Fig. 13 shows the Voila outline, the proof candidate, and

```
1
2 make_atomic
3     using Lock(r, x)
4     with G@r {
5   do {
6
7
8
9     update_region
10        using Lock(r, x) {
11      b := CAS(x, 0, 1)
12    }
13
14
15
16  } inv I while (!b)
17 }
```

```
1 frame Lock(r, x) && G@r {
2   make_atomic
3       using Lock(r, x) with G@r {
4     frame I {
5       do {
6         triple_weak {
7           stabilize {
8             atomic_exists {
9               update_region
10                  using Lock(r, x) {
11                b := CAS(x, 0, 1)
12              }
13            }
14          }
15        }
16      } inv I while (!b)
17    }
18  }
19 }
```

```
1  INFER INTERFERENCE(Lock(r, x))
2    var oldA := r.A
3    exhale r in update
4        && acc(r.A)
5    var oldUpdate := update
6    update := update minus Set(r)
7    exhale acc(r.diamond)
8    unfold Lock(r, x)
9    havoc Lock(r, x)
10
11     b := CAS(x, 0, 1)
12   fold Lock(r, x)
13   UPD TRACK RES(Lock(r, x))
14   update := oldUpdate
15   inhale acc(r.A)
16       && r.A == oldA
17 STABILIZE(Lock(r, x))
```

Fig. 13: Left to right: the core of our running example's `lock` procedure (same as Fig. 3), the proof candidate with inferred bridge rules, and an excerpt of its Viper encoding. Colors link operations of the proof candidate to their encoding. `I` abbreviates the loop invariant from Fig. 3. The encoding uses macros such as `STABILIZE` (more details later in this appendix) to abstract over Viper details.

the Viper encoding (discussed later). We visualize proof candidates by adding steps for *inferred* bridge rule instantiations (e.g. `triple_weak`, denoting TaDA rule AWeakening1), analogous to the user-provided key rule instantiations. For simplicity, some of the inferred steps are omitted.

Changing Triple Kinds. Atomicity changes of a triple are necessary when a non-atomic composite statement has an abstract-atomic sub-statement, such as the `while` statement in Fig. 11 with its atomic body. In such cases, we apply `triple_weak` (line 6 in Fig. 13) to obtain a non-atomic triple from an atomic one. The corresponding TaDA rule AWeakening1 requires that the postcondition is stable, which we achieve via stabilization, that is, by applying a specialized TaDA entailment that weakens the postcondition to satisfy stability constructively. We denote this step with a `stabilize` annotation (line 7) in the proof candidate.

Framing. TaDA's frame rule requires the *frame*, i.e., the assertion preserved across a statement, to be stable. We infer frames greedily, that is, we (actually, Viper) frame as much information around a statement as soundly possible. For simple statements such as heap accesses, this approach automatically leads to stable frames. For composite statements with (arbitrary) user-provided *footprints* (assertions such as loop invariants describing which resources are taken into the composite statement), we need to ensure explicitly that our greedy approach does not produce an unstable frame. For this purpose, we insert explicit `frame` bridge steps (line 4) around composite statements; all other resources are then framed across, and our encoding will ensure that these frames are stable. In our case, such composite statements are loops (invariants), calls (pre- and postconditions) and

`make_atomic` (using-clauses). In each case, a step `frame` F is inserted, indicating that "everything but footprint F" will be framed across and must thus be stable.

Interference Contexts. TaDA's rules for opening a region and calling a procedure (OpenRegion and FunctionCall; both not necessary for our running example) require that the state of each involved region in the precondition is bound by exactly one interference context ($\forall$). This is not guaranteed in arbitrary TaDA proofs (where a region's state might, e.g. not be bound at all), but it is in Voila, due to our normal form. As a consequence, no additional step is necessary before opening a region; before calling a procedure (not used by our running example), a `substitution` step is inserted to check compatibility of the caller's and the callee's interference contexts. However, the frequently necessary atomicity triple changes from non-atomic to atomic triples violate the single binder restriction of our normal form since non-atomic triples have no interference contexts; similarly, opening a region may violate the restriction since the state of nested regions is typically not bound. To re-establish the normal form, we insert `atomic_exists` steps in both cases, which automatically determine suitable interference contexts for unbound region state, e.g. on line 8, where the preceding `triple_weak` changes triple atomicity.

Levels. Region levels have been omitted from the core paper, but are supported by Voila. Levels are essentially an order on region instances, and are used to prevent circular reasoning when nesting TaDA's duplicable regions. When a region is opened or updated, or when a procedure is called, instantiating the corresponding rule requires a specific triple level. E.g. to open a region (see rule OpenRegion from Fig. 6), the current level (conclusion) must be one higher than the level of the region to open. To meet such requirements, we infer suitable instances of AWeakening3, to changes the triple level, for every rule – with specific level requirements – already present in the proof candidate. Inferring and instantiating AWeakening3 is relatively straightforward, and we believe that our heuristic never fails to infer a suitable application, if one is possible.

G Extended Discussion of Validating Proof Candidates in Viper

Recall from Sec. 6 that proof candidates – i.e. the user-provided program with heuristically inserted bridge rule applications – do not necessarily represent valid proofs, and that we check validity of a proof candidate by encoding it into Viper, and verifying the resulting encoding. If the candidate is invalid, the latter will fail. In this section, we provide additional – but still somewhat high-level – details about the encoding of our running example. Later sections of this appendix build on this, and refine the encoding further, to provide more and more technical details.

G.1 Primer on Viper

Viper uses implicit dynamic frames [40], a dialect of separation logic [31] where a points-to assertion such as `x.f` $\mapsto$ `v` is separated into access permission `acc(x.f)` and heap-dependent expressions `x.f == v`. Similarly, a separation logic predicate [32] is typically represented by a Viper predicate that denotes permissions to a data structure, complemented by a heap-dependent mathematical function that abstracts over the values in the data structure.

Viper provides a simple, object-based, imperative language, which includes all statements necessary to represent TaDA programs, and makes this part of the encoding trivial. In addition, Viper provides two statements to manipulate assertions. For an assertion A, `inhale` A adds all permissions denoted A to the current state and assumes all logical constraints in A. Conversely, `exhale` A asserts all logical constraints in A and checks that the permissions in A are available in the current state (verification fails if either check does not succeed). Moreover, it removes these permissions and assigns non-deterministic values to the corresponding memory locations (to reflect that other program components may now hold the permissions and modify the memory locations). In contrast to exhaling A, asserting A only checks that an assertion holds (and fails otherwise), but does not remove permissions.

G.2 Regions and Assertions

TaDA's regions introduce various resources such as region predicates and guards. We encode those into Viper's permissions and predicates as summarized in Fig. 14 (left). Each region `R` gives rise to a predicate with the same name and parameters, 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 since the function may refer to values to which the predicate provides permissions. Moreover, we introduce an abstract Viper predicate `R_g` for each guard `g` of the region; their uniqueness properties are reflected in the encoding of proof steps such as `stabilize` in Fig. 13.

These declarations allow us to encode most TaDA assertions in a fairly straightforward way. For instance, the assertion $\texttt{Lock}_r(\texttt{x}, s)$ from Fig. 11 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. For instance, we use the permission to the `diamond` field to encode the TaDA resource $r \Mapsto \blacklozenge$. Similarly, the permissions to the fields `R_from` and `R_to` represent TaDA's $r \Mapsto (x, y)$ resource, while the fields' values reflect the arguments x and y. Therefore, $r \Mapsto (0, 1)$ from Fig. 11 is encoded as `acc(`r`.Lock_from) && acc(`r`.Lock_to) &&` r`.Lock_from == 0 &&` r`.Lock_to == 1`.

Besides assertions, TaDA judgments include an interference context and an atomicity context. An interference context of the form $\mathbb{\forall} s \in X$, associated with a region `R(`r`, ...)`, is represented by a field r`.R_X`, which stores the set of values

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

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

foreach g(p̄': t̄') ∈ G:
  predicate R_g(r: Ref, p̄':⟦t̄'⟧)
end

field diamond: Bool
field R_from: T
field R_to: T

field R_X: Set[T]
field R_A: Set[T]
```

```
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
field Lock_from: Int
field Lock_to: Int

field Lock_X: Set[Int]
field Lock_A: Set[Int]
```

Fig. 14: Repetition of Fig. 4 from Sec. 6, showing the Viper encoding of regions in the general case (left), and for the lock region from Fig. 3 (right). The encoding function is denoted by double square brackets; overlines denote lists. The *foreach* loop is 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).

to which the environment may set the region's abstract state. The encoding of an atomicity context $\mathcal{A}$, which tracks pending updates and prevents multiple such updates for the same region instance, is more involved. As explained in App. E, we check the proof outline for all atomicity contexts within a lower and an upper bound. The lower bound is represented by a set-typed variable `update`, local to each procedure (see Fig. 13); its value is the set of all regions currently being updated. This set is modified by `make_atomic` and read by `update_region`, to account for the side conditions of the corresponding TaDA rules. The upper bound, stored in variable `alevel` (omitted from Fig. 13 for simplicity), is used for verifying procedure calls: specifically, to ensure that there is not already a pending update for a region the callee might update as well.

Theoretically, procedure specifications could include the set of regions the procedure might update, but this would require additional overhead. Moreover, such a specification would in general have to include *all* potentially updated regions, including those nested in other regions (which, for recursively defined regions, could be arbitrarily many). As confirmed by the TaDA authors in personal

communication, TaDA currently does not address this problem in a modular way: instead, when a proved procedure triple is used, the proof tree essentially needs to be inlined at call site, to recheck atomicity context side conditions.

For Voila, we devised a modular solution that piggybacks on TaDA's levels to overapproximate the set of regions a procedure can update: first, we determine the highest level $\lambda_{\max}$ of all regions syntactically occurring in a procedure's precondition; this will be the procedure's level (each TaDA procedure specification triple has one) and the initial upper bound of a procedure body's, stored in `alevel`. Now, due to the level-related side conditions of TaDA's proof rules, the procedure cannot update any region with a level *higher* than $\lambda_{\max}$. During procedure body verification, `alevel` is updated (by `make_atomic`) to always reflect the *lowest* level of any region for which an update is pending. When a call is encountered, it now suffices to check that the caller's current atomicity level (`alevel`) is higher than the callee's level ($\lambda_{\max}$) – this guarantees that the callee will not update any region for which an update is already pending.

Lastly, the domain of an update $\mathcal{A}(\mathtt{r})$ is encoded with a set-typed field `r.R_A`. Its value influences assertion stabilization: while an update is pending (i.e., inside `make_atomic`), the environment may not take the region value out of `r.R_A`; the latter is set to `r`'s interference context (`r.R_X`) when `make_atomic` is entered.

G.3 Rule Applications

Recall from Sec. 6 that our proof candidates are tree structures (analogous to proof trees in standard Hoare logic), and that we check the validity of a proof candidate by checking the validity of each rule application in it. For that, we (among other things) check that the necessary triple preconditions hold, and that the executed code establishes the necessary postconditions.

Example. We illustrate our encoding scheme on the body of the loop in our running example, see Fig. 13. We discuss the proof top-down in the Hoare logic, that is, inside-out in the proof candidate and Viper encoding, starting with the `CAS` statement. The `CAS` statement itself is encoded as a Viper method whose specification provides the semantics of the operation.

The proof candidate wraps the `CAS` statement inside an application of the UpdateRegion rule (the blue part in the middle column of Fig. 13; the rule itself is shown in Fig. 2). Lines 2-6 of the Viper encoding (right column) deal with the atomicity context; we omit a detailed explanation for brevity, but recall App. F. The subsequent `exhale` and `unfold` encode steps 1 and 2 of the rule application: instead of exhaling the entire precondition of the conclusion (step 1) and inhaling the precondition of the premise (step 2), the encoding represents only the *net effect* of these two operations. Therefore, it exhales the diamond resource $r \Mapsto \blacklozenge$. Going from the conclusion to the premise, UpdateRegion replaces the region predicate (here, `Lock(r,x)`) by its interpretation. Given the region encoding from Fig. 14, this is exactly what Viper's unfold operation does. Note that we instantiate the conjunct $P(x)$ in the UpdateRegion rule to represent all other resources and properties that hold in the prestate of the rule application. Hence,

it does not show up in the encoding. The subsequent `havoc` operation assigns a non-deterministic value to the state of all held, *still folded* `Lock(r,x)` predicates. This step is necessary because TaDA region predicates are duplicable. $P(x)$ thus could contain such predicate instances (in addition to the unfolded one), and we must prevent Viper from using those instances to frame old region state around the `CAS` statement, which would be unsound. As confirmed by the authors in personal communication, the latter problem is actually currently not addressed in TaDA.

The first two Viper statements after the `CAS` statement (right column, lines 12-13) encode steps 3 and 4 of the rule application: the fold operation replaces the interpretation of the `Lock` predicate by the predicate itself. `UPD_TRACK_RES` is an encoding macro (macro definitions are shown in App. K), which inhales, depending on the success of the `CAS` operation, one of the tracking resources $r \Mapsto (0,1)$ or $r \Mapsto \blacklozenge$. Analogously to $P(x)$ in the precondition, we take Q_1 and Q_2 to represent all other resources and properties that hold in the poststate of the rule application in these two cases. Since they occur in both postconditions, there is no net effect of inhaling and then exhaling them, and we can omit them from the encoding. The final two instructions (lines 14-16) in the blue part of the encoding maintain the atomicity context.

Besides UpdateRegion, the loop body contains three additional rule applications. `atomic_exists` (green section) establishes the interference context, which we encode via macro `INFER_INTERFERENCE`. `triple_weak` (orange) weakens an atomic triple in its premise to a non-atomic triple in its conclusion. Since our encoding does not track the triple kind explicitly, `triple_weak` is not directly reflected in the encoding. However, its conclusion – like all non-atomic triples – must be stable. This side condition is enforced in the encoding via the `STABILIZE` macro. We explain both stability and our treatment of interference contexts next.

Stability and Interference Context Inference. 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 actively *stabilizing* assertions in the Viper encoding. That is, we remove information from 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.

Fig. 15 shows the encoding of stabilization for instances of our `Lock` region (macro `STABILIZE`). First, the region state is havocked, i.e., all information about the state is thrown away. Afterwards, the new region state is assumed to be any state reachable by the environment from the old state. We encode this property of reachability by the environment in two steps: `ENV_MAY_HOLD` yields whether a guard may be held by the environment. The encoding depends on the guard kind:

```
INTERFERENCE_PERMITTED(Lock(r, x), from, to) ≜
     (none < perm(r.diamond) ==> Lock_State(r, x) in r.Lock_A)
  && (   from == 0 && to == 1 && ENV_MAY_HOLD(Lock_G(r))
      || from == 1 && to == 0 && ENV_MAY_HOLD(Lock_G(r)) )

ENV_MAY_HOLD(Lock_G(r)) ≜ perm(Lock_G(r)) == none

STABILIZE(Lock(r, x)) ≜
  label pre_havoc
  havoc Lock(r, x)
  inhale INTERFERENCE_PERMITTED(Lock(r, x),
      old[pre_havoc](Lock_State(r, x)), Lock_State(r, x))

INFER_INTERFERENCE(Lock(r, x)) ≜
  havoc r.Lock_X
  inhale forall s: Int :: s in r.Lock_X
     <==> INTERFERENCE_PERMITTED(Lock(r, x), Lock_State(r, x), s)
```

Fig. 15: Encoding of stabilization and interference inference for the `Lock` example. Viper labels enable referring to the verification state at a particular point in the program (i.e., they generalize `old` expressions, which refer to the prestate of a method). We assume that symbols introduced by macros, e.g. label `pre_havoc`, are always fresh and never result in name clashes. The Viper expression `perm(`ρ`)` denotes the permission currently held to a resource ρ.

the environment can hold the unique guard `G` only if it is not already present in the proof state. In contrast, duplicable guards may always be held by the environment, in which case `ENV_MAY_HOLD` would be defined as `true`. Building on `ENV_MAY_HOLD`, `INTERFERENCE_PERMITTED` encodes the actual reachability property: the environment may perform a state transition if it holds at least the guard that is required for this transition by the transition system. Furthermore, the transition has to stay within the atomicity context if an update is still pending, which is TaDA's interference rely-guarantee. To avoid computing the transitive closure, Voila requires (and checks) transition systems to be transitively closed.

The encoded reachability (macro `INTERFERENCE_PERMITTED`) is also essential for the inference of interference contexts. Intuitively, the smallest interference context, at a given program point, corresponds to the set of states that the environment could transition to, which is exactly the set we already need for stabilization. Therefore, as shown in macro `INFER_INTERFERENCE`, we can obtain a suitable interference context by constraining `r.Lock_X` to be the set of all states reachable by the environment.

G.4 Application of Built-in Viper Rules

Viper provides and automates several structural proof rules, especially the rule of consequence and the frame rule. Soundness of our encoding requires that these Viper rules are used only where permitted by TaDA.

TaDA's entailment rule requires entailments to be justified by view shifts [7,36], whereas Viper's rule of consequence may be applied for any valid entailment.

We must, thus, ensure that Viper's entailment steps are indeed permitted by TaDA's entailment rule. This is the case because TaDA's view shifts impose extra requirements only on entailments that involve region and guard assertions, which are encoded as predicates in Viper. Since Viper does not automatically (un)fold predicate instances, it cannot automatically establish entailments between region assertions. Similarly for guards: encoded as abstract predicates, Viper treats them as uninterpreted resources from which no additional information can be deduced.

Viper automatically frames information about its verification state around all statements. To ensure soundness, we explicitly remove information from the state that would otherwise be framed unsoundly, as we have illustrated with the `havoc` operation in Fig. 13.

H Encoding a Custom Guard Algebra

Voila provides a high degree of automation, as demonstrated by our evaluation in Sec. 7. For concepts not directly supported and automated, it provides various features, such as ghost code, to encode them manually. Crucially, all of these features operate on the level of Voila; programmers do not need to understand (or even be aware of) the encoding into Viper. In this section, we demonstrate Voila's support for manual encodings by an example that uses a custom guard algebra.

Specifically, we chose a TaDA-adaptation [36] of Owicki-Gries' classical parallel-increment example: given multiple threads that successively increment a shared counter in parallel, prove that the final counter state equals the sum of the local increments. To achieve the latter, the TaDA proof uses the custom guard algebra defined in Fig. 16, which defines resources (as guards) for tracking increments, and laws that govern their use and allow relating local and total increments. The example is included in our evaluation (`CounterCl`), and, to the best of our knowledge, cannot be encoded in any comparable tool.

Fig. 17 shows the Voila declaration of region `CClient`, whose manipulation is governed by aforementioned guard algebra. Guards `INC` and `TOTAL` are declared as `manual` to indicate that they are not part of a guard algebra that Voila automates (see also App. B). In particular, this means that Voila will not make any uniqueness assumptions about these guards, e.g. when stabilizing region state. The laws of the guard algebra are encoded as lemma procedures such as `INC_split`, which encodes the left-to-right direction of definition 1. Region `CClient` abstracts over the shared `Counter(r,n,x)`, whose value `n` corresponds to the total increment count; guard `G`, declared by region `Counter` (see Fig. 19), is needed to increment that value. The region's `actions` clause demonstrates Voila's most general syntax for specifying region transitions, and declares that the region state can be incremented from any `n` to any larger `m`, by anybody holding a non-zero fraction of `INC` (regardless of the latter's local increments value `k`). Fig. 17 also shows the specification of procedure `single_client`, whose implementation (shown in Fig. 19) loops until it made `v` successive increments to the shared counter. Note that `single_client`

$$\textsc{Inc}(n_1 + n_2, \pi_1 + \pi_2) = \textsc{Inc}(n_1, \pi_1) \bullet \textsc{Inc}(n_2, \pi_2) \tag{1}$$

$$\textsc{Total}(m) \bullet \textsc{Inc}(n, 1) \implies n = m \tag{2}$$

$$\textsc{Total}(m) \bullet \textsc{Inc}(n, \pi) = \textsc{Total}(m + d) \bullet \textsc{Inc}(n + d, \pi) \tag{3}$$

Fig. 16: Custom guard algebra (an instance of Iris' authoritative monoid [18]) used by the TaDA adaptation of Owicki-Gries' classical parallel-increment example. Guard Inc counts local increments, and can be split and merged, similar to fractional permissions [3], in which case the local increments are split/merged as well. Guard Total, in contrast, is exclusive and counts the overall increments. Composing the whole Inc instance with Total allows concluding that the sum of the local increments equals the total count. Lastly, both values can only be changed in lockstep.

could be parametric in the permission amount required for `INC` (currently fixed to `1/2`), which would allow arbitrarily many parallel instances (e.g. `1/t` for a statically-unknown number of `t` threads).

Fig. 18 shows the central part of the verified code: first, guard `INC(0,0)` is split into two equal fractions by using lemma procedure `INC_split`; afterwards, two calls to `single_client` are run in parallel. Upon termination, lemma procedure `INC_merge` (whose straightforward declaration we omitted), corresponding to the right-to-left direction of guard algebra definition 1, is used to combine the `INC` guards obtained from the postconditions of `single_client` into a single instance `INC(20,1f)`. Subsequent ghost code then opens (unfolds) region `CClient` to bind the – at this point unknown – value of the counter to the logical variable `n`. Finally, lemma procedure `TOTAL_INC_equality`, corresponding to guard algebra definition 2, is used to learn that `n`'s value is equal to `INC`'s value, i.e. `20`. Note that the lemma application would (here) fail for values other than `20`, and that it is possible to work with statically unknown values, e.g. `m1`, `m2` and `m1 + m2` instead of constants `9`, `11` and `20`.

In addition to lemma procedures, Voila provides several ghost operations for manipulating its verification state, including: in-/exhale statements for gaining/giving up resources; unfold/fold statements for opening/closing regions; but also region ghost fields, e.g. for witnessing existentials. All of these can be used to encode TaDA proof steps that are beyond what Voila automates, and to experiment with potential extensions. Ghost operations are always applied on the Voila level such that users do not need to be aware of the encoding into Viper.

```
region CClient(id s, id r, cell x)
  guards {
    manual INC(int, frac);
    manual TOTAL(int);
  }
  interpretation {
    Counter(r, x, ?n) && G@r && TOTAL(n)@s
  }
  state { n }
  actions {
    ?n, ?m, ?k, ?p | 0f < p && n < m | INC(k, p): n ~> m;
  }

lemma INC_split(id s, int k1, int k2, frac p1, frac p2)
  requires INC(k1 + k2, p1 + p2)@s;
  requires 0f < p1 && 0f < p2;
  ensures  INC(k1, p1)@s && INC(k2, p2)@s;

procedure single_client(id s, id r, cell x, int m)
  requires CClient(s, r, x, _) && INC(0, 1/2)@s;
  ensures  CClient(s, r, x, _) && INC(m, 1/2)@s;
```

Fig. 17: Example declarations from the Voila encoding of TaDA's counter-client example, including the `CClient` region, and the signature of procedure `single_client` (see also Fig. 19), which is executed by each thread. Lemma procedure `INC_split` encodes the left-to-right direction of definition 1 from Fig. 16 by means of pre- and postconditions. The remaining algebra laws are encoded analogously, and omitted for brevity.

```
// Allocate memory and create region instances ...

use INC_split(s, 0, 0, 1/2, 1/2);

parallel {
  single_client(s, r, x, 9);
  single_client(s, r, x, 11);
}

use INC_merge(s, 9, 11, 1/2, 1/2);

unfold CClient(s, r, x);
assert Counter(r, x, ?n);

use TOTAL_INC_equality(s, n, 20);
assert n == 20;

// ... destroy region instances and deallocate memory
```

Fig. 18: The central part of the Voila encoding of TaDA's Owicki-Gries adaptation: We use a lemma method to split the guard `INC` before the parallel execution of two calls to `single_client`. After the calls, another lemma method is used to recombine `INC` and to sum up the local increments. Finally, we assert the equality between local and total increments.

```
struct cell {
  int f;
}

region Counter(id r, cell x)
  guards { unique G; }
  interpretation { x.f |-> ?n }
  state { n }
  actions { ?n, ?m | n < m | G: n ~> m; }

abstract_atomic procedure incr(id r, cell x)
  interference ?n in Int;
  requires Counter(r, x, n) && G@r;
  ensures Counter(r, x, n + 1) && G@r;

procedure single_client(id s, id r, cell x, int m)
  requires CClient(s, r, x, _) && INC(0, 1/2)@s;
  ensures  CClient(s, r, x, _) && INC(m, 1/2)@s;
{
  int i := 0;

  while (i < m)
    invariant CClient(s, r, x, _);
    invariant INC(i, 1/2)@s;
  {
    use_atomic
      using CClient(s, r, x, ?v) with INC(i, 1/2)@s;
    {
      incr(r, x);
      use TOTAL_INC_inc(s, v, i, 1/2);
    }

    i := i + 1;
  }
}
```

Fig. 19: Further Voila code from the parallel counter example: the `Counter` region and its `incr` procedure, and the implementation of the `single_client` procedure that is executed by each thread. We slightly simplified the loop invariant by omitting obvious properties. The body of `incr`, omitted for brevity, is similar to procedure `lock` from our running example in Fig. 3: a loop around a `CAS` that attempts to increment the counter by one.

I Soundness

In this section, we briefly explain how to show that our approach is sound w.r.t. TaDA. A comprehensive proof sketch is available in App. L. Recall our high-level approach: we take a Voila procedure proof outline with precondition, body, and postcondition, expand it into a proof candidate by adding further rule applications, encode the proof candidate into Viper, and verify the resulting Viper program. To prove soundness, we need to show that for each proof outline successfully verified in this approach, there exists a derivation in TaDA for a TaDA triple whose precondition, statement, and postcondition correspond to those in Voila. For this proof, we assume that the Viper verification backend verifiers are sound; showing their soundness is an orthogonal concern.

We establish soundness in two main steps: we first prove a lemma that relates the execution of Viper statements and states to TaDA judgments and derivations. In a second step, we instantiate this lemma to show that, for a verified Voila procedure, there indeed exists a corresponding TaDA derivation.

Intuitively, our lemma expresses the following property: given a Viper prestate, a sequence of Viper statements corresponding to an encoded Voila statement, and the Viper poststate determined by executing the sequence of Viper statements, we can derive a valid TaDA triple. The proof goes by induction on the (program and rule) statements of the proof candidate. It maps Viper states to TaDA assertions, and uses the statements and rule applications in the proof candidate to construct a TaDA proof. To enable the mapping from Viper states to TaDA assertions, we prove that our encoding maintains several invariants on Viper states. Some of these invariants are due to Voila's normal form, for instance, that Viper states correspond to stable TaDA assertions for pre- and non-atomic postconditions. Others are due to global properties of TaDA: e.g. that a region under update (diamond resource is held) is always in the current atomicity context, which is a prerequisite for a successful TaDA proof. Yet other invariants are technicalities enabling the mapping from Viper states to TaDA assertions, such as having either no or full permission (rather than arbitrary fractions) to certain Viper fields and predicates.

The above lemma relates the Viper encoding to a derivation of a TaDA triple. What remains to be shown is that this triple actually corresponds to the Voila procedure we encoded. For the triple's precondition and statement, this correspondence is ensured by construction, since we obtain them from the Voila proof candidate. The latter also implies that the initial Viper state (corresponding to the Voila precondition) satisfies aforementioned invariants. For the triple's postcondition (and to conclude the soundness proof), we need to show that the user-provided Voila postcondition is implied by the TaDA postcondition that we obtained from the final Viper state. This is also ensured by construction, since the last Viper statement asserts the Voila postcondition.

J Macro Definitions for our Running Example

In App. G, an overview of our encoding was given, which utilized macros to structure the encoding, and to abstract over the generated Viper code. If a verification backend other than Viper were to be chosen, the macro definitions would most likely have to be adapted, but (probably) not how these macros are combined.

In this section, we present the definitions of several core macros, i.e. the Viper code they expand to: macros `ACTION_PERMITTED` and `LESS` are concerned with checking that a state transition is valid and enabled by held guards; `INTERFERENCE_PERMITTED` and `STABILIZE` simulate environment interference and ensure stable assertion, respectively; and `INFER_INTERFERENCE` is crucial for reducing user-required annotations, by inferring all internal interference contexts.

The definitions shown in this section have been instantiated for our running example (the lock region), and the corresponding explanations refer to the running example to build up intuition. Subsequently, section App. K show all macros, in their general form.

J.1 Transition System Compliance

Recall that MakeAtomic (cf. Fig. 2 and Fig. 6; likewise for UseAtomic) requires checking that a region state change is permitted by the region's transition system, using a particular guard; in our example, the update from 0 to 1 (and vice versa) using guard G. In general, checking compliance of a state change requires showing that there exists a region transition (1) that can be instantiated such that its pre- and poststate match the performed state change, and (2) that is enabled by a specific guard (the one specified in the proof outline).

Fig. 20 shows how macro `ACTION_PERMITTED` encodes these two requirements (as previously mentioned, the shown macro definition is specific to the running example's `Lock` region): since the number of transition options (`actions`) is always finite, a disjunction of the different options suffices. A transition option with specified guard g' is enabled by a guard g if, according to the guard algebra, guard g entails g'; this is encoded by macro `LESS`. Encoding this guard entailment for the algebra of guard G from our running example is straightforward: G is entailed by itself, potentially combined with other guards (e.g. in larger examples). In general, the definition of `LESS` is more involved since Voila supports more complex guard algebras (recall Fig. 7), but the general encoding is similar: e.g. given a fractional guard algebra, g entails g' if g's fraction is larger.

J.2 Environment Interference and Assertion Stability

Recall from App. G that verifier state is stabilized by simulating possible transitions that the environment is permitted to perform. This simulation is a constructive approach to satisfying TaDA's assertion stability requirement: an assertion A is stable if, for each region instance R and for each guard (in general, combination of guards) g potentially held by the environment, A does not contradict R being

```
ACTION_PERMITTED(Lock, from, to, g) ≜
      from == to
   || from == 0 && to == 1 && LESS(Lock_G, g)
   || from == 1 && to == 0 && LESS(Lock_G, g)

LESS(Lock_G, Lock_G && _) ≜ true
LESS(_     , _          ) ≜ false
```

Fig. 20: Definition of macros `ACTION_PERMITTED` and `LESS`, instantiated for our running example. For simplicity, macro definitions utilize structural pattern matching as known from, e.g. Haskell. `LESS` encodes guard entailment, i.e. `LESS(`g`,` g'`)` is true if, according to the guard algebra, guard g' entails g. For brevity, the definition is shown modulo commutativity of guard composition (i.e. the case for `_ && G` is omitted).

in any state reachable with g. Our constructive approach eliminates the need for higher-order quantifications over region instances and guards – which are typically not supported by automated verification backends.

Fig. 21 shows the definition of macro `STABILIZE`, which encodes assertion stabilization, and three helper macros: (1) `INTERFERENCE_PERMITTED` is similar to `ACTION_PERMITTED`, and states that the environment is permitted to perform state transitions if it *could* hold the necessary guards. (2) Correspondingly, `ENV_MAY_HOLD` encodes if the environment could hold certain guards: e.g. only if not held by the current context, for unique guards such as G; and always, for duplicable guards (not used here). The Viper expression `perm(`ρ`)` denotes the permission amount currently held to a resource ρ. (3) Finally, `STABILIZE` stabilizes verification state by first havocking a region's state, followed by constraining it to be any state reachable (by the environment) from the pre-havoc state. To avoid a fixpoint computation, Voila requires (and checks) that state transition systems are transitively closed.

The first line of `INTERFERENCE_PERMITTED` accounts for the TaDA's property that, while an update is pending (i.e. before the linearization point is reached), the environment may not take a region `r` outside the current procedure's interference context `r.X`. More details about the latter are provided in subsection App. J.3.

Lastly, note that any assertion can be checked for stability by inhaling it, stabilizing it, and asserting it; this is done by Voila for region interpretations, procedure specifications and loop invariants, all of which must be stable.

J.3 Interference Context Inference

In TaDA, every rule is parametrized with an interference context (denoted by X in the proof rules, see e.g. Fig. 6) for *atomic* triples, but not for *non-atomic* ones. As a consequence, when going from non-atomic triples to atomic triples, e.g. when sequentially composing atomic statements, a potentially different interference context is newly required.

```
INTERFERENCE_PERMITTED(Lock(r, x), from, to) ≜
     (none < perm(r.diamond) ==> Lock_State(r, x) in r.Lock_A)
  && (   from == 0 && to == 1 && ENV_MAY_HOLD(Lock_G(r))
      || from == 1 && to == 0 && ENV_MAY_HOLD(Lock_G(r)) )

ENV_MAY_HOLD(Lock_G(r)) ≜ perm(Lock_G(r)) == none

STABILIZE(Lock(r, x)) ≜
  label pre_havoc
  havoc Lock(r, x)
  inhale INTERFERENCE_PERMITTED(Lock(r, x),
           old[pre_havoc](Lock_State(r, x)), Lock_State(r, x))
```

Fig. 21: Encoding of stabilization, split into three macros. Viper labels enable referring to the verification state at a particular point in the program (i.e. they generalize `old` expressions, which refer to the state in which the precondition held). The Viper expression `perm(`ρ`)` denotes the permission amount currently held to a resource ρ, here to predicate instance `Lock_G(r)`.

In Voila, users only need to specify interference contexts once (as part of a procedure's signature), whereas all other interference contexts are inferred, via macro `INFER_INTERFERENCE`. More specifically, we infer the smallest interference context (at a given program point) that accounts for all possible environment transitions – which is exactly the set we already need for stabilizing Viper's verification state. Consequently, macro `INFER_INTERFERENCE`, shown in Fig. 22, determines a lock's interference context by first havocking the corresponding field and then constraining the context to exactly those states that the environment could reach.

Note that TaDA in principle allows arbitrarily small interference contexts, but we have not yet found an example where our inference heuristic prevented a successful verification. Furthermore, note that initial interference contexts from procedure preconditions still influence (in particular, restrict) inferred contexts, but only indirectly, via the encoding of MakeAtomic.

In addition to inferring intermediate interference contexts, Voila also automatically propagates interference contexts to nested regions (region assertions occurring in another region's interpretation), which are not even visible in procedure specifications. To illustrate how interference contexts are propagated to nested regions, consider a double counter region `DCounter(r, x, y)` whose interpretation contains two counters `Counter(r1, x)` and `Counter(r2, y)`, and whose region state is the sum of the individual counter states. When opening the `DCounter`, we constrain the interference context of counters `r1` and `r2` to contain value s_1 and s_2, respectively, iff `DCounter`'s interference context contains $s_1 + s_2$. This approach generalizes straightforwardly to more complex situations.

```
INFER_INTERFERENCE(Lock(r, x))
  havoc r.Lock_X
  inhale forall s: Int :: s in r.Lock_X
     <==> INTERFERENCE_PERMITTED(Lock(r, x), Lock_State(r, x), s)
```

Fig. 22: Viper encoding of interference context inference: a region's interference context `r.Lock_X` is inferred to be the set of states the environment could currently reach.

K General Macro Definitions

This section presents all encoding macros, in their general form; we suggest to read App. J first, to build up an intuition for the encoding. The macros are also referenced from the the soundness sketch shown in App. L.

```
ACTION_PERMITTED(R, from, to, g) ≜
  from == to || (exists A in ACTIONS(R) :
    exists x̄_A :: from == from_A(x̄_A) && to == to_A(x̄_A)
      && c_A(x̄_A) && LESS(g_A(x̄_A), g)
  )

INTERFERENCE_PERMITTED(R(r, p̄), from, to) ≜
       (none < perm(r.diamond) ==> to in r.R_A)
    && from == to || (exists A in ACTIONS(R) :
          exists x̄_A :: from == from_A(x̄_A) && to == to_A(x̄_A)
            && c_A(x̄_A) && ENV_MAY_HOLD(g_A(x̄_A))
       )

LESS(g, g') ≜ ...

ENV_MAY_HOLD(g) ≜ ...
```

where R denotes a region name (e.g. Lock), and R(r, $\overline{p}$) a region instance with identifier r and remaining arguments $\overline{p}$, and where $\mathtt{from}_A(\overline{x_A})$ denotes the expression $\mathtt{from}_A$, but with $\overline{x_A}$ substituted for the free (quantified) variables that occur in $\mathtt{from}_A$

Fig. 23: General definitions of macros ACTION_PERMITTED and INTERFERENCE_PERMITTED (whereas the versions shown in Fig. 20 were instantiated for our running example). ACTION_PERMITTED encodes if a transition is valid, given a specific guard; INTERFERENCE_PERMITTED encodes interference the environment could cause. Macros LESS (is a guard entailed by another?) and ENV_MAY_HOLD (could the environment hold a particular guard?) are specific per supported guard algebra, and have been omitted for brevity. ACTIONS(R) denotes the finite set of actions (transitions) declared by region R. Macro function "*exists* A *in* ACTIONS(R) : E(A)" expands to an iterated disjunction $\mathtt{E}(\mathtt{A}_0)$ || $\mathtt{E}(\mathtt{A}_1)$ || .... Types (e.g. for $\overline{x_A}$) have been omitted for brevity, they can be unambiguously inferred from, e.g. involved regions. Given an action A, the expressions c_A, g_A, $\mathtt{from}_A$ and $\mathtt{to}_A$ denote the four components an action declaration comprises.

```
STABILIZE(R) ≜
  label pre_stabilize
  havoc forall r, p̄ :: none < perm(R(r, p̄)) ==> R(r, p̄)
  inhale forall r, p̄ :: none < perm(R(r, p̄)) ==>
    INTERFERENCE_PERMITTED(
      R(r, p̄),
      old[pre_stabilize](R_state(r, p̄)), R_state(r, p̄))

INFER_INTERFERENCE(R) ≜
  havoc forall r :: r != null ==> r.R_X
  inhale forall r, p̄, s :: none < perm(R(r, p̄)) ==> (
    s in r.R_X <==> INTERFERENCE_PERMITTED(R(r, p̄), R_state(r, p̄), s)
  )

LINK_INTERFERENCE(R(r, p̄), s) ≜
  label pre_link
  havoc (forall c in C : c.R_c_X)
  inhale forall m̄_C ::
    (forall c in C : m_c in c.R_c_X)
      <==>
    (StateFunction(R(r, p̄), m̄_C) in r.R_X)
  s
  havoc (forall c in C : c.R_c_X)
  inhale (forall c in C : c.R_c_X == old[pre_link](c.R_c_X))
```

where C is the finite set of region identifiers (e.g. `r'`) that occur in the interpretation of `R(r, p̄)`, and R_c is the region name associated with region identifier `c`

Fig. 24: General definitions of macros `STABILIZE` and `INFER_INTERFERENCE` (whereas the versions shown in Fig. 15 where instantiated for our running example), and of `LINK_INTERFERENCE`. `STABILIZE` accounts for potential environment interference and ensures that only stable facts can be deduced. `INFER_INTERFERENCE` infers interference contexts and `LINK_INTERFERENCE` binds the interference contexts of regions nested in another region instance's interpretation. Intuitively, `LINK_INTERFERENCE` propagates constraints on a nesting region's interference contexts to the interference contexts of the nested regions. Recall that `r.R_X` (e.g. `r.Lock_X`) is the interference context of an instance of a region `R` with identifier `r`. Macro function "*forall* `c` *in* C : $E(c)$" expands to an iterated conjunction `E(c₀) && E(c₁) && ...`. Similarly, $\overline{m_C}$ expands to m_{c_0}, m_{c_1}, ..., one variable m_c for each $c \in C$. StateFunction(`R(r, p̄)`, $\overline{m_C}$) denotes the state of `R(r, p̄)` where the state of each region $c \in C$ occurring in the region interpretation is substituted by m_c. E.g. consider a region `Sum(r, p̄)` with an interpretation `Cell(`c_1`, `$\overline{p_1}$`, ?a) && Cell(`c_2`, `$\overline{p_2}$`, ?b)` and the state clause `a + b`. Then, StateFunction(`Sum(r, p̄)`, m_{c_1}, m_{c_2}) is m_{c_1} `+` m_{c_2}. For this `Sum` example, the first `inhale forall` in the definition of `LINK_INTERFERENCE` would be instantiated as `forall` m_{c_1}, m_{c_2} `::` (m_{c_1} `in` c_1`.Cell_X && `m_{c_2} `in` c_2`.Cell_X) <==> ((`m_{c_1} `+` m_{c_2}`) in r.Sum_X)`.

```
ATOMIC(s) ≜
  label pre_atomic
  foreach R in Regions do
    INFER_INTERFERENCE(R)
  end
  s
  foreach R in Regions do
    havoc forall r :: r != null ==> r.R_X
    inhale forall r :: r != null ==> r.R_X == old[pre_atomic](r.R_X)
    STABILIZE(R)
  end
```

Fig. 25: General definition of macro `ATOMIC`, which encodes changing a non-atomic to an atomic triple and establishing the interference context in accordance with our normal form. Macro function "*foreach* `R` *in* Regions *do* `S(R)` *end*" expands to an iterated sequential composition of Viper statements `S(`R_0`); S(`R_1`); ...`, and Regions denotes the finite set of regions declared by the current Voila program.

```
CALL(y̅ := M(e̅)) ≜
  label pre_call
  foreach l in Levels(M) do
    assert level > l && alevel > l
  end
  var z̅ := e̅
  exhale Pre_M[z̅/x̅]
  foreach R in Regions do
    STABILIZE(R)
  end
  havoc y̅
  inhale Post_M[z̅/x̅][y̅/r̅][old[pre_call]/old]

CALL_ATOMIC(y̅ := M(e̅)) ≜
  label pre_call
  foreach Q in Inter(M) do
    assert r_Q.R_Q_X subset S_Q
  end
  foreach l in Levels(M) do
    assert level > l && alevel > l
  end
  var z̅ := e̅
  exhale Pre_M[z̅/x̅]
  foreach R in Regions do
    STABILIZE(R)
  end
  havoc y̅
  inhale Post_M[z̅/x̅][y̅/r̅][old[pre_call]/old]
```

where $\overline{x}$ and $\overline{r}$ are procedure `M`'s formal in- and out-arguments, respectively, and where $e[a/b]$ denotes syntactic substitution of a with b in e

Fig. 26: General definitions of macros `CALL` and `CALL_ATOMIC`. `Pre_M` and `Post_M` denote `M`'s pre- and postcondition, respectively. Viper variables `level` and `alevel` track the current judgment and atomicity level, respectively. Levels(`M`) denotes the set of all levels that (directly) occur in the precondition of procedure `M`; they effectively determine the level of the procedure to be called, and thus must be smaller than the current levels. Inter(`M`) denotes the set of interference clauses of procedure `M`. Set $\mathtt{S}_Q$ denotes the interference set itself (e.g. `Set(0, 1)` in our running example), and $\mathtt{R}_Q$ and $\mathtt{r}_Q$ denote the region name and identifier (e.g. `Lock` and `r`) that identify the constrained region instance, respectively. Calls to non-atomic procedures are encoded in the expected way, aside from the levels check and the stabilization of the frame. Invocations of atomic procedures are encoded analogously, with the additional check that the caller's interference contexts may not allow more interference than the callee permits.

```
UPDATE_REGION(R(r, l, p̄), s) ≜
  label pre_update

  assert level > l
  var level_store := level
  level := l

  exhale r in update && acc(r.R_A)
  var update_store := update
  update := update minus Set(r)

  exhale acc(r.diamond)

  unfold R(r, l, p̄)
  havoc R(r, l, p̄) // Havoc other instances possibly held
  LINK_INTERFERENCE(R(r, l, p̄), s)
  fold R(r, l, p̄)

  // Note: the following if-else statement is abbreviated as
  // UPD_TRACK_RES in Fig. 13
  if (R_state(r, l, p̄) == old[pre_update](R_state(r, l, p̄))) {
    inhale acc(r.diamond)
  } else {
    inhale acc(r.R_from) && r.R_from == old[pre_update](R_state(r, l, p̄))
    inhale acc(r.R_to) && r.R_to == R_state(r, l, p̄)
  }

  update := update_store
  inhale acc(r.R_A) && r.R_A == old[pre_update](r.R_A)

  level := level_store
```

Fig. 27: General definition of macro `UPDATE_REGION`. Statement `s` is executed as part of the expansion of `LINK_INTERFERENCE`. Since we are interested in the updated region's level, the pattern match in the macro's signature is `R(r, l,` $\overline{p}$`)`, i.e. the level `l` is split off from the remaining arguments $\overline{p}$ (analogous to the region identifier `r`). Viper variable `update` tracks the set of region identifiers for which an atomic update is pending, and Viper fields `r.R_from` and `r.R_to` record the performed update; see also macro `MAKE_ATOMIC` in Fig. 32. The `if-else` statement heuristically resolves an angelic choice, which is not supported by Viper: a region update is assumed to have happened if the region state changed. See also Sec. 7. Recall that `r.R_A` is the domain of the atomicity context for a region `R` with identifier `r`.

```
OPEN_REGION(R(r, l, p̄), s) ≜
  label pre_open

  assert level > l
  var level_store := level
  level := l

  unfold R(r, l, p̄)
  havoc R(r, l, p̄) // Havoc other instances possibly held
  LINK_INTERFERENCE(R(r, l, p̄), s)
  fold R(r, l, p̄)

  assert R_state(r, l, p̄) == old[pre_open](R_state(r, l, p̄))

  level := level_store
```

Fig. 28: General definition of macro `OPEN_REGION`. Statement `s` is executed as part of the expansion of `LINK_INTERFERENCE`. The definition is similar to `UPDATE_REGION`, but the last `assert` statement checks that the region state was not changed by executing `s`.

```
USE_ATOMIC(R(r, l, p̄), g, s) ≜
  label pre_atomic

  assert g
  assert R(r, l, p̄)
  assert alevel > l

  assert level > l
  var level_store := level
  level := l

  unfold R(r, l, p̄)
  havoc R(r, l, p̄) // Havoc other instances possibly held
  LINK_INTERFERENCE(R(r, l, p̄), s)
  fold R(r, l, p̄)

  ACTION_PERMITTED(R, R_state(r, l, p̄), old[pre_open](R_state(r, l, p̄), g)

  level := level_store
```

Fig. 29: General definition of macro `USE_ATOMIC`. Statement `s` is executed as part of the expansion of `LINK_INTERFERENCE`. The definition is similar to `UPDATE_REGION` and `OPEN_REGION`, but here, validity of the atomic update performed by `s` is checked.

```
DO_WHILE(s, b, I) ≜
  s
  WHILE(b, I, s)

WHILE(b, I, s) ≜
  label pre_while
  exhale I
  foreach R in Regions do
    STABILIZE(R)
  end
  inhale I

  var oldUpdate := update
  var oldLevel := level
  var oldALevel := alevel
  while (b)
    invariant I
    invariant update == oldUpdate && level == oldLevel
        && alevel == oldALevel
    invariant forall r ::
        r in update ==> acc(r.A) && r.A == old[pre_while](r.A)
    foreach R in Regions do
      invariant forall r :: r != null ==>
          acc(r.R_X) && r.R_X == old[pre_while](r.R_X)
    end
  {
    s
  }
```

Fig. 30: General definitions of macros `DO_WHILE` and `WHILE`. The latter is encoded using a corresponding Viper loop, preceded by an explicit stabilization of the loop's frame. The additional invariants enforce that triple level, atomicity context and interference context are preserved inside the loop.

```
EXPLICIT_FRAME_OUT ≜
  foreach R in Regions do
    exhale forall r, p̄ :: acc(R(r, p̄), perm(R(r, p̄))
  end
  foreach G in Guards do
    exhale forall r, p̄ :: acc(G(r, p̄), perm(G(r, p̄))
  end
  foreach f in Fields do
    exhale forall x :: x != null ==> acc(x.f, perm(x.f))
  end

EXPLICIT_FRAME_IN(lbl) ≜
  foreach R in Regions do
    exhale forall r, p̄ :: acc(R(r, p̄), perm[lbl](R(r, p̄))
  end
  ...
```

Fig. 31: General definitions for macros `EXPLICIT_FRAME_OUT` and `EXPLICIT_FRAME_IN`. The former exhales permissions to all region instances, guards and fields that the Voila program declares and to which the current verification state holds permissions. The later is analogous, but inhales permissions relative to a given label. In Viper, accessibility predicate `acc(x.f)` denotes full (i.e. write) permission to field `x.f`, and is syntactic sugar for `acc(x.f, write)`. Viper also supports fractional permissions [3]; for such a permission π, the syntax is `acc(x.f,`π`)`. Consequently, the last exhale in the definition of `EXPLICIT_FRAME_OUT` instructs Viper to exhale all permission held to a field `x.f`. Predicates are supported analogously, but with additional syntactic sugar: the `acc` around a predicate can be omitted, and `R(x)` (for some predicate `R`) abbreviates `acc(R(x))`, and thus `acc(R(x), write)`.

```
MAKE_ATOMIC(R(r, l, p̄), g, s) ≜
  label pre_atomic

  exhale g
  exhale R(r, l, p̄)
  foreach R in Regions do
    STABILIZE(R)
  end

  label pre_frame
  EXPLICIT_FRAME_OUT

  assert alevel > l
  var alevel_store := alevel
  alevel := l

  assert !(r in update)
  var update_store := update
  inhale acc(r.R_A) && r.R_A == r.R_X
  update := update union Set(r)

  inhale R(r, l, p̄) && R_state(r, l, p̄) in r.R_A
  inhale acc(r.diamond)

  s

  ACTION_PERMITTED(R, r.R_from, r.R_to, g)

  EXPLICIT_FRAME_OUT

  inhale R(r, l, p̄) && (R_state(r, l, p̄) == r.R_to
  inhale old[pre_atomic](R_state(r, l, p̄)) == r.R_from
  exhale acc(r.R_from) && acc(r.R_to)
  inhale g

  update := update_store
  exhale acc(r.R_A)

  alevel := alevel_store

  EXPLICIT_FRAME_IN(pre_frame)
```

Fig. 32: General definition of macro `MAKE_ATOMIC`. Before statement `s` (which is to be proven abstractly atomic) is executed, all regions are stabilized and all other resources are framed out, levels and contexts are adjusted, and the diamond resource is obtained. After the execution of `s`, validity of the performed atomic update is checked, and parts of the pre-state are restored.

```
PROCEDURE(M(p̄) returns (r̄), s) ≜
  method M(p̄) returns (r̄) {
    inhale Pre_M

    foreach R in Regions do
      inhale forall r :: r != null ==> acc(r.R_X)
    end

    var level: Int
    foreach l in Levels(M) do
      inhale level > l
    end
    var alevel: Int := level
    var update: Set[Ref] := Set()

    s

    exhale Post_M
  }

ATOMIC_PROCEDURE(M(p̄) returns (r̄), s) ≜
  method M(p̄) returns (r̄) {
    inhale Pre_M

    foreach R in Regions do
      inhale forall r :: r != null ==> acc(r.R_X)
    end
    foreach Q in Inter(M) do
      inhale r_Q.R_Q_X subset S_Q
    end

    var level: Int
    foreach l in Levels(M) do
      inhale level > l
    end
    var alevel: Int := level
    var update: Set[Ref] := Set()

    s

    exhale Post_M
  }
```

Fig. 33: General definition of macro `PROCEDURE` and `ATOMIC_PROCEDURE`, which are used to encode, and thus prove, procedure specifications. For a non-atomic procedure, `PROCEDURE` first inhales the precondition. Next, necessary resources are inhaled, and local variables declared and constrained. Afterwards, the encoded procedure body is executed. Finally, the postcondition is exhaled. `ATOMIC_PROCEDURE` expands similarly, but for atomic procedures and their interference clauses, denoted by Inter. See also `CALL` and `ATOMIC_CALL` in Fig. 26.

L Soundness

In this section, we present a soundness argument for our Voila encoding. Our encoding is sound when the successful verification of an encoded Voila procedure proof outline implies that the corresponding TaDA procedure satisfies its TaDA specification. We deduce the latter by showing that the procedure specifications are indeed derivable in TaDA.

We argue soundness of our encoding in four steps: first, we determine invariants on Viper's pre- and post-verification states of encoded Voila outline statements (programming language statements and key rules statements). Second, we define a *judgment mapping*, which maps from a pair (υ, s) of Viper verification states υ, satisfying our invariants, and Voila outline statements s to a TaDA judgment. Third, under the assumption of successful verification, we show by structural induction over Voila outline statements that the judgment mapping maps to derivable TaDA judgments. Fourth, we show for each encoded Voila procedure that the judgment mapping, applied to the encoded procedure body and a Viper state satisfying the procedure's precondition, maps to the desired TaDA judgment. Combining these ingredients, we formally connect verification of an encoded proof outline to derivability of a TaDA proof, resulting in the soundness of our encoding.

For a better overview, we first illustrate our approach in more detail on a simplified version of TaDA. Afterwards, we instantiate our approach for normal TaDA. We demonstrate our soundness argument on four particularly challenging steps of our encoding: the handling of calls, triple changes, `make_atomic`, and `update_region`.

L.1 Approach

For the sake of simplicity, before targeting full TaDA, we introduce our approach informally on a simplified version of TaDA. For this simplified version, assume that TaDA judgments are standard Hoare triples of the form $\vdash \{P\}\,\hat{s}\,\{Q\}$, where P, $\hat{s}$, and Q are the precondition, triple statement, and postcondition, respectively. We omit atomic triples, levels, atomicity contexts, interference contexts, and the requirement that pre- or postconditions are stable. We use $\llcorner s \lrcorner$ to reduce a Voila outline statement s to its underlying program statement, by stripping away potentially surrounding rule statements. E.g. the outline statement `update_region using ... { b := CAS(x,0,1) }` is reduced to `b := CAS(x,0,1)`.

To prove soundness, we need a formal connection between an encoded Voila procedure (that successfully verified in Viper) and a TaDA proof. On the Viper side, we have the state of a Viper program, i.e. the verification state, and the encoding of procedures and outline statements. Conversely, on the side of TaDA, we have syntactic judgments and proof rules. To formally connect both, we define a judgment mapping, a mapping from pairs (υ, s) of Viper verification state υ and Voila outline statement s to syntactic TaDA judgments. For our simplified version of TaDA, we can define such a judgment mapping as follows: assume we have a mapping $\phi(\upsilon)$ from Viper verification state υ to assertions of TaDA. Then, a judgment mapping for a Viper verification state υ and a Voila outline statement

s can be defined as $(\!| \upsilon, s |\!) = \{\phi(\upsilon)\} \llcorner s \lrcorner \{\phi(\upsilon')\}$, where $\upsilon' = \mathsf{post}([\![s]\!], \upsilon)$ is the strongest postcondition verification state of the Viper encoding of s and the verification state υ.

The judgment mapping is only applied to prestates of encoded Voila outline statements because only these states, together with encoded statement and resulting poststate, are formally connected to triples in a TaDA proof. In particular, the mapping is not applied to intermediate verification states of a Viper encoding. We define invariants on Viper prestates so that the judgment mapping has stronger guarantees on the mapped verification states. E.g. TaDA does not allow partial ownership of points-to predicates ($\mathtt{x.f} \mapsto \upsilon$). However, such partial permissions are in general possible in Viper states, making judgment mappings for such states with partial permissions nonsensical. Therefore, for our encoding, we define the invariant that permissions to fields are either full or none. We then have to show that these invariants on a verification state hold, before we use the verification state in a judgment mapping. We use $\mathbb{I}$ to refer to the set of all verification states satisfying these invariants.

Using the judgment mapping, we can verbally state our soundness lemma of the outline statement encoding: *"Under the assumption of successful Viper verification, we show that the judgment mapping maps to derivable TaDA judgments when applied to encoded Voila outline statements and Viper verification states satisfying our state invariants"*. Before we can express this property more formally, we have to define the meaning of a successful Viper verification. A successful verification entails that all verification states of the verified Viper program are *valid*. In Viper, a verification state is valid when it is not a special error state $\lightning$. Therefore, we refine the soundness lemma from above as follows: *"Forall Voila outline statements s and Viper verification states $\upsilon \in \mathbb{I}$, a valid strongest poststate* $\mathsf{post}([\![s]\!], \upsilon) \neq \lightning$ *implies that the mapped judgment $(\!| \upsilon, s |\!)$ is derivable in TaDA and that the poststate satisfies our state invariants* $\mathsf{post}([\![s]\!], \upsilon) \in \mathbb{I}$*"*. We first illustrate the purpose of this lemma and then argue how to prove it.

The lemma aids us to derive that a successfully verified Voila procedure implies that the corresponding TaDA procedure with its specification is derivable: Consider a Voila procedure with the specification $\{P\}\, \mathsf{m}(\dots)\, \{Q\}$ where $\mathsf{m}(\dots)$ is the procedure itself. Let s_m be its body and let υ_{pre} be the verification state before the encoding of its body. If the procedure is encoded as $\mathtt{inhale}\ [\![P]\!];\ [\![s_m]\!];\ \mathtt{exhale}\ [\![Q]\!]$, then $\upsilon_{\mathsf{pre}} = \mathsf{post}(\mathtt{inhale}\ [\![P]\!], \upsilon_{\mathsf{zero}})$, where υ_{zero} is the initial (empty) verification state. Assuming υ_{pre} satisfies our state invariants ($\upsilon_{\mathsf{pre}} \in \mathbb{I}$) and that $(\!| \upsilon_{\mathsf{pre}}, s_m |\!)$ maps to $\{P\}\, s_m\, \{Q'\}$ with $Q' \models Q$, we can apply the lemma to get that $\{P\}\, s_m\, \{Q\}$, and as such $\{P\}\, \mathsf{m}(\dots)\, \{Q\}$, is derivable in TaDA.

We can prove soundness of the outline statement encoding by straightforward structural induction over outline statements. We illustrate a case of the induction at an abstract level. Consider a compound outline statement $s\{s'\}$ (s is the compound, e.g. `update_region`, and s' is its body, e.g. `CAS(...)`) with an encoding $[\![s\{s'\}]\!] = \mathtt{c}_1;[\![s']\!];\mathtt{c}_2$, where $\mathtt{c}_1$ and $\mathtt{c}_2$ are the Viper statements before and after the encoding of the body, respectively. There are four Viper verification states of interest: the prestate of the compound statement υ_0, the prestate of

its body $\upsilon_1 = \mathsf{post}(\mathsf{c}_1, \upsilon_0,)$, the poststate of its body $\upsilon_2 = \mathsf{post}([\![s']\!], \upsilon_1,)$, and the poststate of the compound statement $\upsilon_3 = \mathsf{post}(\mathsf{c}_2, \upsilon_2,)$. From the induction hypothesis, we know that $(\!| \upsilon_1, s' |\!) = \{\phi(\upsilon_1)\} \llcorner s' \lrcorner \{\phi(\upsilon_2)\}$ is derivable in TaDA and we have to show that $(\!| \upsilon_0, s\{s'\} |\!) = \{\phi(\upsilon_0)\} \llcorner s\{s'\} \lrcorner \{\phi(\upsilon_3)\}$ is derivable in TaDA. Showing this derivation corresponds to applying rules to fill the (?)-gap in the following proof tree:

$$\dfrac{\dfrac{\vdots}{\{\phi(\upsilon_1)\} \llcorner s' \lrcorner \{\phi(\upsilon_2)\}}\,(\mathrm{IH})}{\{\phi(\upsilon_0)\} \llcorner s\{s'\} \lrcorner \{\phi(\upsilon_3)\}}\,(?)$$

The application of IH denotes using the fact from the induction hypothesis that $\{\phi(\upsilon_1)\} \llcorner s' \lrcorner \{\phi(\upsilon_2)\}$ is derivable in TaDA. The necessary rule applications for the (?)-gap are determined by our encoded proof candidate (Sec. 5), where we have to argue that their applications are correctly encoded in the outline statement encoding.

In the next sections, we first discuss Viper's verification state. Afterwards, we introduce the judgment mapping and state invariants for all of TaDA, including atomic triples, levels, atomicity context, interference context, and stability requirements. Last, we argue soundness of the outline statement encoding.

L.2 Viper Verification State

Viper's verification state [31] is defined as a set of traces. Each trace consists of a sequence of *state atoms*: a triple (H, P, S) of a heap H (mapping `Ref` and field name pairs, as well as applied functions, to values), a permission mask P (mapping `Ref` and field name pairs, as well as predicate instances, to permission amounts; these amounts are non-negative rationals, which for fields cannot exceed 1), and a variable store S (mapping variables to values). Furthermore, a trace consists of a label mapping lbl, mapping Viper labels to their corresponding state atom. We have a special error verification state $\lightning$, which is the result of a verification error, e.g. the poststate of `x := 5; assert x == 4`. We use υ to range over verification states. The semantics of the core logic is given in [31]. In particular, the semantics of heap-dependent expressions such as fields accesses `x.f` comes with well-definedness conditions. E.g. reading from a field is only allowed in states with a non-zero permission for that field. The semantics of functions and predicates follows [41].

When a Viper program verifies successfully, this implies that all `assert` and `exhale` (removes the assertion from the verification state, introduced in Sec. 6) assert and exhale, respectively, assertions valid in there respective verification states. This includes implicit assertions and exhales, such as asserting non-zero permission when accessing a field or exhaling preconditions when calling functions.

L.3 TaDA Judgment Mapping and State Invariants

In the judgment mapping of TaDA, we distinguish between non-atomic and abstract atomic Voila outline statements. For an abstract atomic outline statement s_{a} and a Viper prestate υ, the judgment mapping maps to a TaDA judgment of the following shape:

$$\begin{aligned}(\!|\, \upsilon, s_{\mathsf{a}} \,|\!) = \forall \overline{h} \in H_\upsilon.\ \forall \mathcal{A} \text{ with } \mathcal{A}_\upsilon^{\mathsf{lb}} \leq \mathcal{A} \leq \mathcal{A}_\upsilon^{\mathsf{ub}}.& \\ \lambda_\upsilon; \mathcal{A} \ \vdash\ \mathbb{W}\overline{x} \in X_\upsilon \left\langle\ \mathsf{Pre}_\upsilon(\overline{h}, \overline{x})\ \right\rangle \llcorner s_{\mathsf{a}} \lrcorner \left\langle\ \mathsf{Post}_{\upsilon,\upsilon'}(\overline{h}, \overline{x})\ \right\rangle& \\ \text{where } \upsilon' = \mathsf{post}([\![\, s_{\mathsf{a}} \,]\!], \upsilon)&\end{aligned}$$

For the sake of brevity, we omit the exact judgment mapping definition; instead, we describe the different components informally: **(H_υ)** Viper can deduce facts based on knowledge of old state, e.g. from the fact that some variable had the value 5 at a previous Viper label. To account for such deductions at the TaDA level, we use a set H_υ, the *history set*, to capture Viper's knowledge about old state. E.g. consider a variable `z` whose value is one plus its old value from label `lbl`. With the history set, this fact is captured as $\forall(..., h_{\mathtt{z}}, ...) \in H_\upsilon.\, \mathtt{z} = h_{\mathtt{z}} + 1$, where $h_{\mathtt{z}}$ binds the part of the history set that tracks `z`'s value from label `lbl`. In our TaDA judgment, we do not map these facts to the pre- or postcondition of a TaDA triple because rules such as MakeAtomic restricts the shape of pre- and postconditions. This would force us to remove these facts from TaDA's pre- and postconditions, even though these facts remain in Viper's verification state. **($\mathcal{A}$)** As described in App. E, a TaDA triple is proven forall atomicity contexts $\mathcal{A}$ within a lower bound $\mathcal{A}_\upsilon^{\mathsf{lb}}$ and an upper bound $\mathcal{A}_\upsilon^{\mathsf{ub}}$; the order on atomicity contexts is defined as follows:

$$\begin{aligned}\mathcal{A}_1 \leq \mathcal{A}_2 \Leftrightarrow \forall r \in \mathsf{dom}(\mathcal{A}_1).\ r \in \mathsf{dom}(\mathcal{A}_2)& \\ \wedge\ \mathsf{dom}(\mathcal{A}_2(r)) \subseteq \mathsf{dom}(\mathcal{A}_1(r))& \\ \wedge\ \forall z \in \mathsf{dom}(\mathcal{A}_2(r)).\ \mathsf{img}(\mathcal{A}_1(r)(z)) \subseteq \mathsf{img}(\mathcal{A}_2(r)(z))&\end{aligned}$$

The operations $\mathsf{dom}(\cdot)$ and $\mathsf{img}(\cdot)$ denote domain and image, respectively. We define the order such that a Viper assertion P being stable for an atomicity context $\mathcal{A}_1$ implies that P is also stable for all atomicity contexts $\mathcal{A}_2$ with $\mathcal{A}_1 \leq \mathcal{A}_2$. Therefore, to satisfy stability of a pre or postcondition for all atomicity contexts $\mathcal{A}$ that a triple is proven for ($\mathcal{A}_\upsilon^{\mathsf{lb}} \leq \mathcal{A} \leq \mathcal{A}_\upsilon^{\mathsf{ub}}$), it is sufficient to satisfy stability of the pre or postcondition for $\mathcal{A}_\upsilon^{\mathsf{lb}}$. Regarding the mapping, the lower bound $\mathcal{A}_\upsilon^{\mathsf{lb}}$ has an entry for a region instance r only if, in the verification state υ, r is contained in the value of the `update` variable. The domain of such an atomicity context entry for r is the value of r.`R_A`, where `R` is the region name of r. The image of an entry for r depends on the poststate υ'. If r.`R_from` $= z$ and r.`R_to` $= f(z)$ are held in υ' (this corresponds to $r \mapsto (z, f(z))$ in TaDA), then the image of the entry is defined by the function f, otherwise the image is the empty set $\emptyset$. Conversely, the upper bound $\mathcal{A}_\upsilon^{\mathsf{ub}}$ has an entry for region identifier r only if its region level is at least the value of the `alevel` variable in υ. The

domain for r is $\mathsf{dom}(\mathcal{A}_\upsilon^{\mathsf{lb}})$ if r is an entry of the lower bound $\mathcal{A}_\upsilon^{\mathsf{lb}}$ ($r \in \mathsf{dom}(\mathcal{A}_\upsilon^{\mathsf{lb}})$), otherwise the domain for r in $\mathcal{A}_\upsilon^{\mathsf{ub}}$ is $\emptyset$. Again, the image for r depends on the poststate. If r.`R_from` $= z$ and r.`R_to` $= f(z)$ are held in υ', then the image of the entry is defined by the function f, otherwise the image is the set of all values $\mathbb{U}$. **($\boldsymbol{\lambda_\upsilon}$)** The level of the triple λ_υ is the value of the `level` variable in the prestate υ. **($\boldsymbol{X_\upsilon}$)** The interference context X_υ is the cartesian product of all values of `r.R_X` for which the predicate `R(r,` $\overline{p}$`)` is held in the prestate υ. **($\mathsf{Pre}_\upsilon, \mathsf{Post}_{\upsilon,\upsilon'}$)** The pre and postcondition of the TaDA triple are Pre_υ and $\mathsf{Post}_{\upsilon,\upsilon'}$, respectively. Both can have occurrences of quantifiers bound by the history set and the interference context quantifier. We use different assertion mappings for pre and postconditions because the interference context is handled for each of them differently, as they have different restrictions in our normal form. For the precondition, if a region predicate `R(`r`,`λ`,`$\overline{p}$`)` is held in the prestate υ, then this is mapped to $R_r^\lambda(\overline{p}, x_r)$ where x_r is the interference context quantifier for region identifier r. This way, we guarantee that the state of regions in the precondition is bound by the interference context. For the postcondition, we do not have this requirement. Holding `R(`r`,`λ`,`$\overline{p}$`)` in the poststate υ' is mapped to $R_r^\lambda(\overline{p}, z_r)$ where z_r is a logical variable, additionally introduced for binding the region state. The region state function R_State(r,λ,$\overline{p}$) is mapped to constraints on x_r and z_r for the pre and postcondition, respectively. For all other resources the mapping is the same for pre and postconditions. The mapping for these resources corresponds to the inverse of the encoding: E.g. Holding a guard predicate `R_G(`r`,`$\overline{p}$`)` in a verification state is mapped to a TaDA guard instance $[\mathrm{G}(\overline{p})]_r$. Similarly, holding `acc(x.f)` is mapped to x.f $\mapsto z$ where z is the value of `x.f` in the verification state. For simplicity, we omit assertions with local program variables in the shown TaDA triple. These are mapped to private assertions of atomic triples and can only depend on the history set. No other resource, e.g. guards or points-to predicates, are mapped to private assertions of atomic triples. The handling of local variables in all rule applications is straightforward.

For a non-atomic outline statement s_{na} and a Viper prestate υ, the judgment mapping maps to a non-atomic TaDA judgment of the following shape:

$$
\begin{aligned}
(\!|\, \upsilon, s_{\mathsf{na}} \,|\!) = \forall \overline{h} \in H_\upsilon.\ \forall \mathcal{A} \text{ with } \mathcal{A}_\upsilon^{\mathsf{lb}} \leq \mathcal{A} \leq \mathcal{A}_\upsilon^{\mathsf{ub}}.& \\
\lambda_\upsilon; \mathcal{A} \ \vdash\ \left\{\ \mathsf{P}_\upsilon(\overline{h})\ \right\}\ \llcorner s_{\mathsf{na}} \lrcorner\ \left\{\ \mathsf{P}_{\upsilon'}(\overline{h})\ \right\}& \\
\text{where } \upsilon' = \mathsf{post}([\![\, s_{\mathsf{na}} \,]\!], \upsilon)&
\end{aligned}
$$

The mapping is the same as for abstract atomic outline steps, except that the TaDA judgment has no interference context and thus the same assertion mapping P_υ can be used for both, pre- and postcondition.

To express stability of the pre or postcondition, we additionally define a closed form of the pre and postcondition, which has no free variables. The closed form for Viper prestates υ and abstract atomic statements s_{a}, as well as non-atomic

outline statements s_{na}, are derived from TaDA and defined as follows:

$$\widehat{\mathsf{Pre}_{\upsilon,s_{\mathsf{a}}}} = \forall \overline{h} \in H_\upsilon.\ \exists\, \overline{x} \in X_\upsilon.\ \mathsf{Pre}_\upsilon(\overline{h}, \overline{x})$$
$$\widehat{\mathsf{Post}_{\upsilon,s_{\mathsf{a}}}} = \forall \overline{h} \in H_\upsilon.\ \forall\, \overline{x} \in X_\upsilon.\ \mathsf{Post}_{\upsilon,\upsilon'}(\overline{h}, \overline{x})$$
$$\text{where } \upsilon' = \mathsf{post}(\llbracket\, s_{\mathsf{a}} \,\rrbracket, \upsilon)$$
$$\widehat{\mathsf{Pre}_{\upsilon,s_{\mathsf{na}}}} = \forall \overline{h} \in H_\upsilon.\ \mathsf{P}_\upsilon(\overline{h})$$
$$\widehat{\mathsf{Post}_{\upsilon,s_{\mathsf{na}}}} = \forall \overline{h} \in H_\upsilon.\ \mathsf{P}_{\upsilon'}(\overline{h})$$
$$\text{where } \upsilon' = \mathsf{post}(\llbracket\, s_{\mathsf{na}} \,\rrbracket, \upsilon)$$

For stronger guarantees in the judgment mapping, we have several invariants on Viper prestate of encoded Voila outline statements: (1) Fields, region predicates, and guard predicates have either none or full permissions. In particular, permissions for the interference context field (`R_X`) is always full, and permissions to the two tracking fields `R_from` and `R_to` are either both full or both none. An exception are guard predicates for fractional guards, which are allowed to have partial permissions because their Viper permission amount maps to a TaDA guard argument. (2) If permission to the diamond tracking resource field (`r.diamond`) is held, then r is contained in the set of the `update` variable. (3) For all region identifiers contained in `update`, the region level is at least the value of the `alevel` variable. (4) The other invariants are more technical and required to define the judgment mapping.

Besides invariants on single Viper verification states, we define invariants on pairs of pre and poststates of an encoded outline statement. We use $\mathbb{T}$ to denote the set of verification state pairs that satisfy these invariants. A state pair (υ, υ') is contained in $\mathbb{T}$, when their level, interference context, and both atomicity context bounds are equal in the judgment mapping, i.e. when $\lambda_\upsilon = \lambda_{\upsilon'}$, $X_\upsilon = X_{\upsilon'}$, $\mathcal{A}^{\mathsf{lb}}_\upsilon = \mathcal{A}^{\mathsf{lb}}_{\upsilon'}$, and $\mathcal{A}^{\mathsf{ub}}_\upsilon = \mathcal{A}^{\mathsf{ub}}_{\upsilon'}$ holds. For full TaDA, opposed to the simplified version, we need these additional two-state invariants to guarantee that level, interference context, and atomicity context stay consistent for sequential composition.

L.4 Proof Candidates

As discussed in Sec. L.1, we prove soundness of our outline statement encoding by induction over outline statements. We use the induction predicate W:

$$\begin{aligned} W(s) :\equiv \forall \upsilon \in \mathbb{I}.\ \upsilon' \neq \lightning\ \wedge\ \widehat{\mathsf{Pre}_{\upsilon,s}} \textit{ is stable for } \mathcal{A}^{\mathsf{lb}}_\upsilon \Longrightarrow \\ (\!|\upsilon, s|\!) \textit{ is derivable } \wedge\ \upsilon' \in \mathbb{I}\ \wedge\ (\upsilon, \upsilon') \in \mathbb{T} \\ \wedge\ (s \textit{ is non-atomic } \Longrightarrow \widehat{\mathsf{Post}_{\upsilon,s}} \textit{ is stable for} \mathcal{A}^{\mathsf{lb}}_\upsilon) \\ \text{where } \upsilon' = \mathsf{post}(\llbracket\, s \,\rrbracket, \upsilon) \end{aligned}$$

The additional properties about stability can be included in our invariants on pre and post Viper verification states $\mathbb{T}$ (by making the invariants dependent on the

encoded Voila outline statement). We explicitly state the stability properties for clarity. Our normal form is captured in our soundness argument as a combination of the invariants $\mathbb{I}$ and $\mathbb{T}$, the condition on stability in W, and the shape of TaDA judgments in the image of our judgment mapping.

To streamline the proof argument, we add to Voila an outline statement `atomic{s}`, which changes the atomicity of a triple from non-atomic to atomic. Without this additional outline statement, for cases such as loops, we have to make a case distinction whether the body is abstract atomic or non-atomic. By introducing the outline statement, it is guaranteed that the atomicity of the body is the atomicity of the non-bridge rules' premise.

For the induction proof, we focus on the cases for atomic calls, `atomic`, `update_region`, and `make_atomic`. These are particular challenging steps of our encoding. In our presentation of the induction cases, for the sake of brevity, we reason about Viper code at a higher, more abstract, level to focus on the proofs themselves. In particular, we take as a lemma that the poststate υ' after the macro `STABILIZE` (See Fig. 24) is stable for $\mathcal{A}_{\upsilon}^{\mathsf{lb}}$ when mapped to TaDA, where υ is the prestate of the macro. A proof argument about a similar encoding of stabilization was provided in [9].

Atomic Call. Fig. 34 shows the filled out TaDA proof tree for the encoding of an abstract atomic call $\overline{\mathtt{y}}$ `:=` $\mathtt{M}(\overline{\mathtt{e}})$, where `M`, $\overline{\mathtt{e}}$, and $\overline{\mathtt{y}}$ are the called procedure, the arguments, and the return variables, respectively. The encoding of atomic calls is given in Fig. 26. Let υ and υ' be Viper's pre and poststate of the encoded Voila statement, respectively. As seen in the definition of the judgment mapping, the TaDA judgment is proven for every $\overline{h} \in H_{\upsilon}$ and every atomicity context $\mathcal{A}$ between $\mathcal{A}_{\upsilon}^{\mathsf{lb}}$ and $\mathcal{A}_{\upsilon}^{\mathsf{ub}}$. The important steps of the proof snippet go as follows (from the bottom of the tree to the top): Firstly, the current judgment level λ_{υ} is reduced to the level of the called procedure, denoted by λ'. The side condition of AWeakening3 ($\lambda_{\upsilon} \geq \lambda'$) is satisfied, since in the encoding we assert that the `level` variable is larger than every level in `M`'s precondition and as such is larger than λ', which is one plus the maximum level in `M`'s precondition. Secondly, the mapped verification state that is not part of the procedure's precondition $R(\overline{h}, \overline{x})$, is weakened to a stabilized version $R'(\overline{h}, \overline{x})$ (by Consequence), and then framed off. The stability of the frame $R'(\overline{h}, \overline{x})$ is a consequence from the use of the macro `STABILIZE` in the encoding. Furthermore, we know that only the TaDA pre and postcondition of the procedure remain in the proof state since their Viper counterparts are asserted and everything else is framed off. We denote the procedure's pre and postcondition with $P'(\overline{h}, \overline{x})$ and $Q'(\overline{h}, \overline{x})$, respectively. Thirdly, the current interference context X_{υ} is widened to the interference context of the procedure, denoted as X', by applying Substitution. This widening is justified since in the encoding we assert that X_{υ} is a subset of X' for the relevant interference context parts. Lastly, we apply the call rule. We already know that the level, interference context, and pre- and postcondition match. It remains to argue that the current atomicity context $\mathcal{A}$ is contained in the set of atomicity contexts handled by `M`, i.e. that $\mathcal{A}$ is between the lower and upper bound of `M` as defined by the judgment mapping, which we denote by $\mathcal{A}_{\mathsf{M}}^{\mathsf{lb}}$ and $\mathcal{A}_{\mathsf{M}}^{\mathsf{ub}}$, respectively.

The inclusion of the lower bound is trivial since $\mathcal{A}^{\mathsf{lb}}_{\mathtt{M}}$ is empty. The upper bound is satisfied since in the encoding we check that the value of the `alevel` variable is larger or equal to the `M`'s level, which is equal to the the value of `alevel` initially set for `M`, hence entailing $\mathcal{A}^{\mathsf{ub}}_{\upsilon} \leq \mathcal{A}^{\mathsf{ub}}_{\mathtt{M}}$.

Non-atomic calls are similar, except that interference contexts are not present.

$$
\begin{array}{c}
\vdots \\
\hline
\lambda'; \mathcal{A} \vdash \mathbb{V}\, \overline{x'} \in X'.\ \langle\ P'(\overline{h}, \overline{x'})[\overline{\mathtt{e}}/\overline{\mathtt{z}}]\ \rangle\ \overline{\mathtt{y}}\ \mathtt{:=}\ \mathtt{M}(\overline{\mathtt{e}})\ \langle\ Q'(\overline{h}, \overline{x'})[\overline{\mathtt{e}}/\overline{\mathtt{z}}][\overline{\mathtt{y}}/\overline{\mathtt{r}}]\ \rangle \quad (\textsc{Call}) \\
\hline
\lambda'; \mathcal{A} \vdash \mathbb{V}\, \overline{x} \in X_{\upsilon}.\ \langle\ P'(\overline{h}, \overline{x})[\overline{\mathtt{e}}/\overline{\mathtt{z}}]\ \rangle\ \overline{\mathtt{y}}\ \mathtt{:=}\ \mathtt{M}(\overline{\mathtt{e}})\ \langle\ Q'(\overline{h}, \overline{x})[\overline{\mathtt{e}}/\overline{\mathtt{z}}][\overline{\mathtt{y}}/\overline{\mathtt{r}}]\ \rangle \quad (\textsc{Subst}) \\
\hline
\lambda'; \mathcal{A} \vdash \mathbb{V}\, \overline{x} \in X_{\upsilon}.\ \langle\ R'(\overline{h}, \overline{x}) * P'(\overline{h}, \overline{x})[\overline{\mathtt{e}}/\overline{\mathtt{z}}]\ \rangle\ \overline{\mathtt{y}}\ \mathtt{:=}\ \mathtt{M}(\overline{\mathtt{e}})\ \langle\ R'(\overline{h}, \overline{x}) * Q'(\overline{h}, \overline{x})[\overline{\mathtt{e}}/\overline{\mathtt{z}}][\overline{\mathtt{y}}/\overline{\mathtt{r}}]\ \rangle \quad (\textsc{Frame}) \\
\hline
\lambda'; \mathcal{A} \vdash \mathbb{V}\, \overline{x} \in X_{\upsilon}.\ \langle\ R(\overline{h}, \overline{x}) * P'(\overline{h}, \overline{x})[\overline{\mathtt{e}}/\overline{\mathtt{z}}]\ \rangle\ \overline{\mathtt{y}}\ \mathtt{:=}\ \mathtt{M}(\overline{\mathtt{e}})\ \langle\ R'(\overline{h}, \overline{x}) * Q'(\overline{h}, \overline{x})[\overline{\mathtt{e}}/\overline{\mathtt{z}}][\overline{\mathtt{y}}/\overline{\mathtt{r}}]\ \rangle \quad (\textsc{Cons}) \\
\hline
\lambda'; \mathcal{A} \vdash \mathbb{V}\, \overline{x} \in X_{\upsilon}.\ \langle\ \mathsf{Pre}_{\upsilon}(\overline{h}, \overline{x})\ \rangle\ \overline{\mathtt{y}}\ \mathtt{:=}\ \mathtt{M}(\overline{\mathtt{e}})\ \langle\ \mathsf{Post}_{\upsilon,\upsilon'}(\overline{h}, \overline{x})\ \rangle \quad (\textsc{Cons}) \\
\hline
\lambda_{\upsilon}; \mathcal{A} \vdash \mathbb{V}\, \overline{x} \in X_{\upsilon}.\ \langle\ \mathsf{Pre}_{\upsilon}(\overline{h}, \overline{x})\ \rangle\ \overline{\mathtt{y}}\ \mathtt{:=}\ \mathtt{M}(\overline{\mathtt{e}})\ \langle\ \mathsf{Post}_{\upsilon,\upsilon'}(\overline{h}, \overline{x})\ \rangle \quad (\textsc{AWeak3})
\end{array}
$$

Fig. 34: Proof snippet for the encoding of abstract atomic calls $\overline{\mathtt{y}}$ `:=` $\mathtt{M}(\overline{\mathtt{e}})$, where `M`, $\overline{\mathtt{e}}$, and $\overline{\mathtt{y}}$ are the called procedure, the arguments, and the return variables, respectively. The encoding of atomic calls is given in Fig. 26. Note that we use shortened TaDA rule names.

Atomicity Change. The corresponding proof tree is shown in Fig. 35, where `s'` is the TaDA statement reduced from `atomic{s}` (i.e. `s'` $= \llcorner$`atomic{s}`$\lrcorner$). The encoding of atomicity changes is given in Fig. 25. Let υ_0 and υ_3 be Viper's pre and poststate of the encoded Voila statement `atomic{s}`, respectively. Similarly, let υ_1 and υ_2 be Viper's pre and poststate of the encoded Voila statement `s`, respectively. Again, let $\overline{h} \in H_{\upsilon_0}$ and an atomicity context $\mathcal{A}$ between $\mathcal{A}^{\mathsf{lb}}_{\upsilon_0}$ and $\mathcal{A}^{\mathsf{ub}}_{\upsilon_0}$ be arbitrary. As seen in Sec. 5, in TaDA, the atomicity of the triple is changed by applying Consequence to stabilize the postcondition, AWeakening1 to switch the triple kind, AExists to establish the interference context, where $\overline{x} \in X_{\upsilon_1}$ binds all region states in $P'(\overline{h}, \overline{x})$ and the corresponding region states from the linearization point in $Q'(\overline{h}, \overline{x})$. As described in the definition of our judgment mapping, Exists is applied to move Viper facts about old state out of the triple. Afterwards, the induction hypothesis can be applied.

Update-Region. The corresponding proof tree is shown in Fig. 36, where again `s'` is the reduced Viper statement (`s'` $= \llcorner$`update_region using ... {s}`$\lrcorner$). Again, let υ_0 and υ_3 be Viper's pre and poststate of the encoded Voila statement `update_region using ... {s}`, respectively. Furthermore, let υ_1 and υ_2 be Viper's pre and poststate of the encoded Voila statement `s`, respectively. Let $\overline{h} \in H_{\upsilon_0}$ and an atomicity context $\mathcal{A}$ between $\mathcal{A}^{\mathsf{lb}}_{\upsilon_0}$ and $\mathcal{A}^{\mathsf{ub}}_{\upsilon_0}$ be arbitrary. The encoding of `update_region` is given in Fig. 27. The important steps of the proof snippet go as follows (from the bottom of the snippet to the top): Firstly, as for the atomic call, the judgment level is reduced. We denote the new level λ_{υ_1} as λ to not clutter the

$$
\begin{array}{c}
\vdots \\
\hline
\lambda_{\upsilon_1}; \mathcal{A} \ \vdash \ \mathbb{V}\, \overline{x} \in X_{\upsilon_1}.\ \langle\ \mathsf{Pre}_{\upsilon_1}(\overline{h}, \overline{h''}, \overline{x})\ \rangle\ \mathsf{s}'\ \langle\ \mathsf{Post}_{\upsilon_1,\upsilon_2}(\overline{h}, \overline{h''}, \overline{x})\ \rangle \ (\text{IH}) \\
\hline
\lambda_{\upsilon_1}; \mathcal{A} \ \vdash \ \mathbb{V}\, \overline{x} \in X_{\upsilon_1}.\ \langle\ \exists \overline{h''} \in H''.\ \mathsf{Pre}_{\upsilon_1}(\overline{h}, \overline{h''}, \overline{x})\ \rangle\ \mathsf{s}'\ \langle\ \exists \overline{h''} \in H''.\ \mathsf{Post}_{\upsilon_1,\upsilon_2}(\overline{h}, \overline{h''}, \overline{x})\ \rangle \ (\text{Exists}) \\
\hline
\lambda_{\upsilon_0}; \mathcal{A} \ \vdash \ \mathbb{V}\, \overline{x} \in X_{\upsilon_0}.\ \langle\ P'(\overline{h}, \overline{x})\ \rangle\ \mathsf{s}'\ \langle\ Q'(\overline{h}, \overline{x})\ \rangle \ (\text{Cons}) \\
\hline
\lambda_{\upsilon_0}; \mathcal{A} \ \vdash \ \langle\ \exists \overline{x} \in X_{\upsilon_0}.\ P'(\overline{h}, \overline{x})\ \rangle\ \mathsf{s}'\ \langle\ \exists \overline{x} \in X_{\upsilon_0}.\ Q'(\overline{h}, \overline{x})\ \rangle \ (\text{AExists}) \\
\hline
\lambda_{\upsilon_0}; \mathcal{A} \ \vdash \ \{\ \exists \overline{x} \in X_{\upsilon_0}.\ P'(\overline{h}, \overline{x})\ \}\ \mathsf{s}'\ \{\ \exists \overline{x} \in X_{\upsilon_0}.\ Q'(\overline{h}, \overline{x})\ \} \ (\text{AWeak1}) \\
\hline
\lambda_{\upsilon_0}; \mathcal{A} \ \vdash \ \{\ \mathsf{P}_{\upsilon_0}(\overline{h})\ \}\ \mathsf{s}'\ \{\ \mathsf{P}_{\upsilon_3}(\overline{h})\ \} \ (\text{Cons})
\end{array}
$$

Fig. 35: Proof snippet for the encoding of `atomic`, which switches from the non-atomic triples to the atomic triples. The statement s' is equal to ⌞`atomic{s}`⌟. The encoding of atomicity changes is given in Fig. 25. Note that we use shortened TaDA rule names.

proof tree with subscripts. Again, the encoding asserts explicitly that the new level $(\lambda + 1)$ is smaller or equal to the current level (λ_{υ_0}). Secondly, as also seen before, Consequence is used to get the pre- and postcondition into the right shape such that UpdateRegion can be applied next. All specified resources are justified since their encoding is explicitly asserted in the encoding. In the updated region $\mathsf{R}_r^{\lambda}(\overline{p}, x^{\circ})$, we use x° to denote the region's interference context quantifier from the sequence of all interference context quantifiers $\overline{x}$. Thirdly, UpdateRegion is applied, where D and I are the domain and image of the atomicity context entry for r, respectively. Splitting the atomicity context is justified, because the encoding tests explicitly that an entry for r exists in the atomicity context. Recall from the judgment mapping, that we define images of atomicity context entries such that they coincide with the target of the tracking resource $r \mapsto (x^{\circ}, w)$. Therefore, W and I agree on whether or not an update happened. In the encoding, the region instance is opened by unfolding the region predicate, which matches the definition of our resource mapping. Fourthly, as discussed in Sec. 5, the interference contexts of regions contained in $I(\mathsf{R}_r^{\lambda}(\overline{p}, x^{\circ}))$, denoted as $\overline{X'}$, is added to the current interference context X_{υ_0}. Formally, we entail $Z(r, \lambda, \overline{p}, x^{\circ}, \overline{x'})$, which denotes the assertion that is equivalent to $I(\mathsf{R}_r^{\lambda}(\overline{p}, x^{\circ}))$, except that the region state of regions is explicitly bound by $\overline{x'}$. Lastly, as seen for `atomic`, surplus old Viper state is removed by applying Exists, so that the invariant can be applied.

The cases for `open_region` and `use_atomic` are similar.

Make-Atomic. The corresponding proof tree is shown in Fig. 37, where as before s' is the reduced TaDA statement ($\mathsf{s}' =$ ⌞`make_atomic using ... {s}`⌟). Again, let υ_0 and υ_3 be Viper's pre and poststate of the encoded Voila statement `make_atomic using ... {s}`, respectively. Furthermore, let υ_1 and υ_2 be Viper's pre and poststate of the encoded Voila statement `s`, respectively. Let $\overline{h} \in H_{\upsilon_0}$ and an atomicity context $\mathcal{A}$ between $\mathcal{A}^{\mathsf{lb}}_{\upsilon_0}$ and $\mathcal{A}^{\mathsf{ub}}_{\upsilon_0}$ be arbitrary. The encoding of `make_atomic` is given in Fig. 32. The important steps of the proof snippet go as follows (from the bottom of the proof tree to the top):

$$
\begin{array}{c}
\vdots \\
\hline
\lambda; \mathcal{A}' \vdash \mathbb{V}\ (\overline{x}, \overline{x'}) \in X_{v_1}.\ \langle\ \mathsf{Pre}_{v_1}(\overline{h}, \overline{h''}, \overline{x})\ \rangle\ \mathsf{s}'\ \langle\ \mathsf{Post}_{v_1,v_2}(\overline{h}, \overline{h''}, \overline{x})\ \rangle \quad \text{(IH)} \\
\hline
\lambda; \mathcal{A}' \vdash \mathbb{V}\ (\overline{x}, \overline{x'}) \in X_{v_1}.\ \langle\ \exists \overline{h''} \in H''.\ \mathsf{Pre}_{v_1}(\overline{h}, \overline{h''}, \overline{x})\ \rangle\ \mathsf{s}'\ \langle\ \exists \overline{h''} \in H''.\ \mathsf{Post}_{v_1,v_2}(\overline{h}, \overline{h''}, \overline{x})\ \rangle \quad \text{(Exists)} \\
\hline
\lambda; \mathcal{A}' \vdash \mathbb{V}\ \overline{x} \in X_{v_0},\ \overline{x'} \in X'.\ \left\langle\ P'(\overline{h}, \overline{x}) * Z(r, \lambda, \overline{p}, x^\circ, \overline{x'}) * r \mapsto \blacklozenge\ \right\rangle\ \mathsf{s}'\ \left\langle\ \exists w \in W.\ \begin{array}{l}(x^\circ \neq w)\ ?\ I(\mathsf{R}_r^\lambda(\overline{p}, w)) * r \mapsto (x^\circ, w) * Q_1(\overline{h}, \overline{x}, w) \\ :\ Z(r, \lambda, \overline{p}, x^\circ, \overline{x'}) * r \mapsto \blacklozenge * Q_2(\overline{h}, \overline{x})\end{array}\ \right\rangle \quad \text{(Cons)} \\
\hline
\lambda; \mathcal{A}' \vdash \mathbb{V}\ \overline{x} \in X_{v_0}.\ \left\langle\ P'(\overline{h}, \overline{x}) * \exists \overline{x'} \in X'.\ Z(r, \lambda, \overline{p}, x^\circ, \overline{x'}) * r \mapsto \blacklozenge\ \right\rangle\ \mathsf{s}'\ \left\langle\ \exists w \in W.\ \begin{array}{l}(x^\circ \neq w)\ ?\ I(\mathsf{R}_r^\lambda(\overline{p}, w)) * r \mapsto (x^\circ, w) * Q_1(\overline{h}, \overline{x}, w) \\ :\ \exists \overline{x'} \in X'.\ Z(r, \lambda, \overline{p}, x^\circ, \overline{x'}) * r \mapsto \blacklozenge * Q_2(\overline{h}, \overline{x})\end{array}\ \right\rangle \quad \text{(AExists)} \\
\hline
\lambda; \mathcal{A}' \vdash \mathbb{V}\ \overline{x} \in X_{v_0}.\ \left\langle\ P'(\overline{h}, \overline{x}) * I(\mathsf{R}_r^\lambda(\overline{p}, x^\circ)) * r \mapsto \blacklozenge\ \right\rangle\ \mathsf{s}'\ \left\langle\ \exists w \in W.\ \begin{array}{l}(x^\circ \neq w)\ ?\ I(\mathsf{R}_r^\lambda(\overline{p}, w)) * r \mapsto (x^\circ, w) * Q_1(\overline{h}, \overline{x}, w) \\ :\ I(\mathsf{R}_r^\lambda(\overline{p}, x^\circ)) * r \mapsto \blacklozenge * Q_2(\overline{h}, \overline{x})\end{array}\ \right\rangle \quad \text{(Cons)} \\
\hline
\lambda + 1; r : z \in D \to I(z), \mathcal{A}' \vdash \mathbb{V}\ \overline{x} \in X_{v_0}.\ \left\langle\ P'(\overline{h}, \overline{x}) * \mathsf{R}_r^\lambda(\overline{p}, x^\circ) * r \mapsto \blacklozenge\ \right\rangle\ \mathsf{s}'\ \left\langle\ \exists w \in W.\ \begin{array}{l}(x^\circ \neq w)\ ?\ \mathsf{R}_r^\lambda(\overline{p}, w) * r \mapsto (x^\circ, w) * Q_1(\overline{h}, \overline{x}, w) \\ :\ \mathsf{R}_r^\lambda(\overline{p}, x^\circ) * r \mapsto \blacklozenge * Q_2(\overline{h}, \overline{x})\end{array}\ \right\rangle \quad \text{(UpdReg)} \\
\hline
\lambda + 1; \mathcal{A} \vdash \mathbb{V}\ \overline{x} \in X_{v_0}.\ \langle\ \mathsf{Pre}_{v_0}(\overline{h}, \overline{x})\ \rangle\ \mathsf{s}'\ \langle\ \mathsf{Post}_{v_0,v_3}(\overline{h}, \overline{x})\ \rangle \quad \text{(Cons)} \\
\hline
\lambda_{v_0}; \mathcal{A} \vdash \mathbb{V}\ \overline{x} \in X_{v_0}.\ \langle\ \mathsf{Pre}_{v_0}(\overline{h}, \overline{x})\ \rangle\ \mathsf{s}'\ \langle\ \mathsf{Post}_{v_0,v_3}(\overline{h}, \overline{x})\ \rangle \quad \text{(AWeak3)}
\end{array}
$$

Fig. 36: Proof snippet for the encoding of `update_region`, where $\mathsf{R}_r^\lambda(\overline{p}, x^\circ)$ is the updated region. The statement s' is the reduced TaDA statement (⌞`update_region using ... {s}`⌟). The encoding of `update_region` is given in Fig. 27. Note that we use shortened TaDA rule names.

Firstly, similar to calls, the verification state is split into resources required for the `make_atomic` and the frame $R(\overline{h}, \overline{x})$, where again $R'(\overline{h}, \overline{x})$ is the stabilized version that is framed off to the postcondition. Afterwards, MakeAtomic is applied. The new atomicity context for the updated region is $z \in X^{\circ}_{\upsilon_0} \to I(z)$, where $X^{\circ}_{\upsilon_0}$ is the projection of X_{υ_0} onto the interference context for region identifier r. The image I of the atomicity context entry for r is chosen according to our judgment mapping. Similar to the case for `update_region`, I coincides with the target of the tracking resource, thus coincides with W. We can guarantee that r was not in the atomicity context before, since the encoding explicitly checks that r is not in the set of `update` and that the region's level is smaller than the value of the `alevel` variable. Again, old facts are removed by applying Exists. However, before we can use the induction hypothesis, we have to guarantee that the new atomicity context $(r : z \in X^{\circ}_{\upsilon_0} \to I(z), \mathcal{A})$ is between $\mathcal{A}^{\mathsf{lb}}_{\upsilon_1}$ and $\mathcal{A}^{\mathsf{ub}}_{\upsilon_1}$. The lower bound follows straight forwardly from r getting added to the atomicity context and $\mathcal{A}$ being between $\mathcal{A}^{\mathsf{lb}}_{\upsilon_0}$ and $\mathcal{A}^{\mathsf{ub}}_{\upsilon_0}$. The upper bound follows from the value of variable `alevel` in υ_0 being larger then in υ_1, which in the encoding is enforced by first asserting that the new value of `alevel` is lower than the current one and then assigning the new value to the `alevel` variable.

$$
\begin{array}{c}
\vdots \\
\hline
\lambda_{\upsilon_1}; r : z \in X^{\circ}_{\upsilon_0} \to I(z), \mathcal{A} \;\vdash\; \{\; \mathsf{P}_{\upsilon_1}(\overline{h}, \overline{h''}, \overline{x}) \;\}\; \mathsf{s}' \;\{\; \mathsf{P}_{\upsilon_2}(\overline{h}, \overline{h''}, \overline{x}) \;\} \quad \text{(IH)} \\
\hline
\lambda_{\upsilon_1}; r : z \in X^{\circ}_{\upsilon_0} \to I(z), \mathcal{A} \;\vdash\; \{\; \exists \overline{h''} \in H''.\; \mathsf{P}_{\upsilon_1}(\overline{h}, \overline{h''}, \overline{x}) \;\}\; \mathsf{s}' \;\{\; \exists \overline{h''} \in H''.\; \mathsf{P}_{\upsilon_1}(\overline{h}, \overline{h''}, \overline{x}) \;\} \quad \text{(Exists)} \\
\hline
\lambda_{\upsilon_0}; r : z \in X^{\circ}_{\upsilon_0} \to I(z), \mathcal{A} \;\vdash\; \left\{\; \exists x^{\circ} \in X^{\circ}_{\upsilon_0}.\; \mathsf{R}^{\lambda}_{r}(\overline{p}, x^{\circ}) * r \Mapsto \blacklozenge \;\right\}\; \mathsf{s}' \;\{\; \exists x^{\circ} \in X^{\circ}_{\upsilon_0},\; w \in W.\; r \Mapsto (x^{\circ}, w) \;\} \quad \text{(Cons)} \\
\hline
\lambda_{\upsilon_0}; \mathcal{A} \;\vdash\; \mathbb{V}\, \overline{x} \in X_{\upsilon_0}.\; \left\langle\; \mathsf{R}^{\lambda}_{r}(\overline{p}, x^{\circ}) * [\mathrm{G}]_r \;\right\rangle\; \mathsf{s}' \;\left\langle\; \exists w \in W.\; \mathsf{R}^{\lambda}_{r}(\overline{p}, w) * [\mathrm{G}]_r \;\right\rangle \quad \text{(MakeAtomic)} \\
\hline
\lambda_{\upsilon_0}; \mathcal{A} \;\vdash\; \mathbb{V}\, \overline{x} \in X_{\upsilon_0}.\; \left\langle\; R'(\overline{h}, \overline{x}) * \mathsf{R}^{\lambda}_{r}(\overline{p}, x^{\circ}) * [\mathrm{G}]_r \;\right\rangle\; \mathsf{s}' \;\left\langle\; R'(\overline{h}, \overline{x}) * \exists w \in W.\; \mathsf{R}^{\lambda}_{r}(\overline{p}, w) * [\mathrm{G}]_r \;\right\rangle \quad \text{(Frame)} \\
\hline
\lambda_{\upsilon_0}; \mathcal{A} \;\vdash\; \mathbb{V}\, \overline{x} \in X_{\upsilon_0}.\; \left\langle\; R(\overline{h}, \overline{x}) * \mathsf{R}^{\lambda}_{r}(\overline{p}, x^{\circ}) * [\mathrm{G}]_r \;\right\rangle\; \mathsf{s}' \;\left\langle\; R'(\overline{h}, \overline{x}) * \exists w \in W.\; \mathsf{R}^{\lambda}_{r}(\overline{p}, w) * [\mathrm{G}]_r \;\right\rangle \quad \text{(Cons)} \\
\hline
\lambda_{\upsilon_0}; \mathcal{A} \;\vdash\; \mathbb{V}\, \overline{x} \in X_{\upsilon_0}.\; \langle\; \mathsf{Pre}_{\upsilon_0}(\overline{h}, \overline{x}) \;\rangle\; \mathsf{s}' \;\langle\; \mathsf{Post}_{\upsilon_0, \upsilon_3}(\overline{h}, \overline{x}) \;\rangle \quad \text{(Cons)}
\end{array}
$$

Fig. 37: Proof snippet for the encoding of `make_atomic`, where $\mathsf{R}^{\lambda}_{r}(\overline{p}, x^{\circ})$ is the updated region and $[\mathrm{G}]_r$ the for the updated used guard. The statement s' is equal to ⌞`make_atomic using ... {s}`⌟. The encoding of `make_atomic` is given in Fig. 32. Note that we use shortened TaDA rule names.

M Complete Encoding of our Running Example

An overview and excerpt of the encoding of our `lock` running example was shown in Fig. 13; below we show the full encoding atomicity contexts, interference contexts and levels. The encoding uses the macros defined in App. K (see also their example-specific definitions in App. J).

```
method lock(r: Ref, lvl: Int, cell: Ref) {
  // Encoded precondition
  inhale forall c: Ref :: c != null ==> acc(c.Lock_X)
  inhale r.Lock_X == Set(0,1)
  inhale Lock(r, lvl, x) && Lock_state(r, lvl, x) in r.Lock_X
  inhale Lock_G(r)

  // Initialize levels
  var level: Int
  inhale level > lvl
  var alevel: Int := level
  var update: Set[Ref] := Set()

  var b: Bool

  MAKE_ATOMIC(Lock(r, lvl, cell), Lock_G(r), {
    DO_WHILE({
      ATOMIC({
        UPDATE_REGION(Lock(r, lvl, cell), {
          CALL(b := CAS_val(x, 0, 1))
        })
      })
    }, !b, INV)
  })

  // Encoded postcondition
  exhale Lock(r, lvl, x) && Lock_state(r, lvl, x) == 1
  exhale Lock_G(r)
  exhale old(Lock_state(r, lvl, x)) == 0
}

where INV is the encoded source invariant:
  Lock(r, lvl, cell) &&
  (!b ==> acc(r.diamond)) &&
  ( b ==> acc(r.Lock_from) && acc(r.Lock_to) &&
          r.Lock_from == 0 && r.Lock_to == 1)
```

Fig. 38: Viper encoding of procedure `lock` from our running example, with macros not yet expanded.