Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Proving Refinement in a Rust Verifier

Master Thesis

Jan Schär

May 10, 2023

Advisors: Prof. Dr. Peter Müller, Aurel Bílý, João Pereira
Department of Computer Science, ETH Zürich

Abstract

Software verification has the goal of ensuring correctness of software. Refinement is a verification technique, where we prove that one system implements a smaller, simpler system, which serves as a specification for the larger system. The technique is especially useful for the verification of concurrent and distributed systems. We would like to use refinement to prove that a program written in a language such as Rust correctly implements an abstract specification in the style of TLA$^+$.

In this thesis, we present an approach for describing and verifying refinement between a TLA$^+$ model and an implementation written in Rust, based on the approach described in "Flexible Refinement Proofs in Separation Logic" by Bílý et al. [1]. We implemented this approach in Prusti, a verification tool for Rust. The model state is embedded in the implementation as ghost state, protected by a ghost lock. The ghost lock can be acquired to perform steps in the model; this way model steps are linked to implementation steps. Model steps can be protected by a guard; these steps can then exclusively be performed by the thread which owns the guard. For the verification of larger applications, a modular approach is needed. We support trusted modules for primitives such as atomic variables, as well as verified modules with an inner model, which can be linked to the outer model where the module is used. Finally, we demonstrate the approach on two case studies.

Acknowledgements

I thank my supervisors Aurel Bílý and João Pereira for supporting me in this project with their input and feedback. This was an interesting and exciting topic, and I am thankful for the opportunity to work on it.

Contents

Chapter 1

Introduction

Prusti [2] is a tool for the verification of Rust [3] code. It makes use of Rust's ownership system, and can verify general properties such as absence of panics, and also custom specifications given as annotations in the Rust code. This is useful for implementing software correctly.

TLA$^+$ [4] is a language for writing high-level, mathematical models of a system, expressed as a state machine. Modeling a software system in TLA$^+$ is useful for designing it correctly.

In TLA$^+$, a lower-level, more detailed model can be connected through a *refinement* to a higher-level model. The refinement is a theorem which states that the low-level model implements the high-level model. This theorem can be checked using a model checker or proof system.

A TLA$^+$ model is already valuable without formally verifying that the code implements it, as it allows developers to identify design problems in their programs. Nonetheless, programs may still have implementation issues not captured by the model, and they can even fail to comply with the formal model written in TLA$^+$. As such, in this thesis, we explore how to extend existing refinement approaches to the implementation, thus closing the gap between the TLA$^+$ models and the concrete implementation.

More concretely, the goal of this thesis is to develop an approach for describing and verifying refinement between a TLA$^+$ model and an implementation written in Rust, based on the general approach described in "Flexible Refinement Proofs in Separation Logic" by Bílý et al. [1]. We only target safety properties, liveness properties are out of scope.

We implement and evaluate the approach with Prusti. However, we did not extend or modify Prusti itself, and the approach could also be implemented in other verification tools with similar features.

1.1 Outline

As an introduction to writing specifications, we start by specifying a regular lock (Section 3.1).

For the refinement approach, we first need to translate the TLA$^+$ model to Prusti syntax, so that we can refer to it from specification annotations (Section 3.2). We embed the model state as ghost state in the implementation, protected by a ghost lock, which forms the core of the approach (Section 3.3). Guards are ghost resources which can protect certain steps of the model, and can be used in an exclusive or in a shared read-only mode (Section 3.4).

To support verification of concurrent systems, we model atomics in our approach, both with sequentially consistent and with weaker release-acquire ordering (Section 3.5).

To allow splitting large systems into smaller, independently verified parts, and to allow reuse of parts, we introduce modules, which can be used in a model, and can be either trusted or verified (Section 3.6). The atomics form a trusted module. Our hash set case study uses the atomics module, and is itself a verified module, which could be used in a bigger application.

While most conditions for the correctness of the approach are stated as preconditions of the various methods and thus verified automatically, there are some assumptions that need to be checked manually (Section 3.7).

We evaluate our approach on two case studies: A consensus algorithm for distributed systems (Section 4.1), and a concurrent data structure (Section 4.2).

Chapter 2

Background

2.1 Refinement approach

This thesis builds on the general approach to refinement described in "Flexible Refinement Proofs in Separation Logic" by Bílý et al. [1], which we summarize here.

An abstract transition system (ATS) is defined by the variables of the abstract state, and the `Init` and `Next` relations. `Init` defines the initial states, and `Next` defines transitions between states. Listing 2.1 shows an example ATS, written in the TLA$^+$ language. Figure 2.1 shows a trace of this ATS.

We want to prove that a program implements, or refines, the model. Formally, a state machine m refines state machine m', if any trace of m is also a trace of m'. However, we cannot directly compare an execution trace of the program with a trace of the ATS. To do so would require the model to describe the entire state of the program in a detailed operational semantics, which is

EXTENDS *Integers*, *Sequences*
VARIABLES *count*, *stdout*

$$Init \triangleq count = 0 \land stdout = \langle\rangle$$

$$Next \triangleq \land count' = count + 1 \\ \land stdout' = Append(stdout, count)$$

Listing 2.1: Example TLA$^+$ model. In *Next*, unprimed and primed variables refer to the state before and after the transition, respectively.

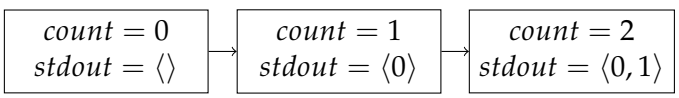

Figure 2.1: A possible trace (sequence of states) of the example.

impractical. Instead, we describe the execution of the program in terms of the model variables.

As a first step, we embed the model variables into the program as *ghost variables*. These ghost variables can be modified in ghost statements. When the program is compiled, ghost variables and statements are removed; they are only there for verification. Because ghost code cannot affect control flow of the program, removing it is safe.

But this is not enough. Note that in the example model, the *count* and *stdout* variables must be updated simultaneously. To achieve this, we introduce a *ghost lock*, which protects all model variables. The variables may only be changed while holding the lock, and we consider all changes to the model variables between acquiring and releasing the lock as one atomic update. With these ghost updates in place, we can now check if the trace of the model variables during the program execution is a trace of the ATS.

However, so far, the approach does not relate the ghost updates to what the actual code is doing. To address this, we represent all interactions of the program with the environment in the model, and make sure that what happens in the model corresponds to what happens in the code. We achieve this by declaring some model variables to be *environment variables*. In the example, *stdout* is an environment variable. Then, we identify all methods of the program which interact with the environment, and specify the corresponding change to the environment variables. In the example, we could have a `print` method, which is specified as appending its argument to the *stdout* sequence. Finally, we disallow changing environment variables in ghost code; they may *only* be changed by calling the methods that actually interact with the environment.

Now that we have defined the refinement, we want to statically verify that all program traces are also ATS traces. First, we need to show that `Init` holds when we create the ghost lock after initializing ghost variables. Then, we have two proof obligations for every block of code that acquires and releases the ghost lock: (1) The `Next` relation must hold for the model state before and after the block. (2) The code executed in the block must be atomic. By discharging these proof obligations, we finally obtain a proof that the program implements the model.

Why do we have to prove atomicity of the code executed while holding the ghost lock? This is because, otherwise, the actions of two threads could interleave, such that the actual behavior of the system no longer corresponds to the behavior of the model. Another way to see this is that the ghost lock is treated as if it was a regular lock during verification, but at runtime, the acquire/release calls are erased, and this erasure is only sound if the code between the calls was already atomic.

For the proof that the `Next` relation is satisfied, we often need to keep some knowledge about the abstract state, without holding the ghost lock. This is achieved by the thread having ownership of (or, in the case of Rust, a mutable reference to) a ghost resource, without which certain transitions in the ATS are not allowed. We can then reason that if a certain property of the abstract state held when we last held the ghost lock, it will still hold when acquiring the ghost lock again. There are different types of ghost resources that can be used here, the approach does not prescribe which is used.

2.2 Lipton's Reduction

The general approach by Bílý et al. [1] requires the code executed in ghost lock steps to be atomic, but leaves open how to check atomicity. In this thesis, we use reduction by Lipton [5] for this.

The question considered by Lipton is: When can we treat a block R in a program P as if it was atomic, when making proofs about P, even though R consists of multiple actions? We define P/R, called the *reduction* of P by R, as the program obtained from P by replacing R with a single indivisible action which has the same effect as R. If R satisfies certain restrictions, then we can prove some property for the simpler program P/R, and obtain the same property for P.

The key idea by Lipton is to define certain atomic actions to be right movers or left movers. f is a *right mover* if, for any computation $\alpha f h$ where α is a computation and f and h are actions in different threads, $\alpha h f$ is also a computation, and the final values of program variables in $\alpha f h$ and $\alpha h f$ are the same. Similarly, g is a *left mover* if, for any computation $\alpha h g$, where h and g are in different threads, $\alpha g h$ is also a computation, and the values of program variables in $\alpha h g$ and $\alpha g h$ are the same.

We define P/R to be a *D-reduction* if R consists of atomic statements $S_1; \ldots; S_k$, and for some i, $S_1, \ldots, S_{i-1}$ are right movers and $S_{i+1}, \ldots, S_k$ are left movers (S_i is unconstrained), and all of $S_2, \ldots, S_k$ can always execute. Lipton then shows that a D-reduction P/R halts if and only if P halts.

Intuitively, a right mover can be moved across all actions happening in other threads, up to the point where the next action is in the same thread as the right mover, and the result of the program is the same. Similarly, left movers can be moved to the left across actions in other threads. If we move all left and right movers as far as possible, the result is that all of $S_1, \ldots, S_k$ of a D-reduction are right next to each other, with no interleaving actions by other threads. And this explains why, even though $S_1, \ldots, S_k$ may be interleaved with other threads, we can treat it as if it was a single atomic action.

2.3 Rust

Rust [3] is a relatively new systems programming language. Unique about Rust are the safety guarantees enabled by its ownership and lifetimes system. The safe subset of Rust is guaranteed to have no memory safety violations or data races.

Each resource has exactly one owner, which can access it. We can *borrow* a resource, which makes the original resource inaccessible for the lifetime of the borrow. Shared borrows are denoted by `&` and allow read-only access to a resource; multiple such borrows can exist simultaneously. Exclusive borrows are denoted by `&mut` and allow modifications to the resource; there can only be one such borrow at a time, and no shared borrows can exist at the same time.

Interior mutability allows us to circumvent these rules; it allows modifications of values behind a shared borrow. For example, a lock can be shared between threads, such that each thread has a shared borrow to the same lock. After acquiring the lock, the value stored inside can be modified.

2.4 Prusti

Prusti [2] is a verification tool for Rust programs. It extends the syntax of Rust with specification annotations. We can then run the verification tool to check whether the specifications of a Rust program are valid. Prusti relies on a powerful SMT solver to perform verification automatically for the most part, but in some situations additional annotations are needed to help the tool find proofs. Here we provide a quick overview; for more details see the Prusti user guide [6].

Prusti defines an extended expression syntax for specifications. In addition to regular Rust expressions, we can write universal and existential quantifiers. There are convenience operators for logical implications. The *snapshot equality* operator can compare two values by their snapshots. For example, the snapshot of a struct contains the values of all fields, but values with interior mutability are not part of the snapshot.

In a Rust method, we can add assert and assume statements, which contain a Boolean specification expression. For each assert, the tool checks that the expression is true, and raises an error if it cannot prove it. For an assume statement on the other hand, it will assume that the expression is true without checking it; it simply trusts the specification author.

For each Rust method, we can add pre- and postconditions. Preconditions are specification expressions that are asserted to be true where the method is called, and which we assume are true at the start of the method body.

Postconditions on the other hand are asserted before the method returns, and assumed after the call to the method. In postconditions, we can use `old(...)` expressions to refer to the value of parameters at the time when the method was called, and we can use the special variable `result` to refer to the returned value.

Loops can generally not be verified automatically, therefore we need to provide loop invariants. These are specification expressions which must hold before the loop starts, and must be preserved by the loop body.

We differentiate between methods and functions. *Methods* can have side effects and arbitrary control flow, and may or may not return a value. *Functions* on the other hand behave like mathematical functions, they have no side effects and always return the same value for the same arguments. In Prusti, functions are annotated with the `pure` attribute. In specification expressions, we can only call functions, but not methods. Functions can also have pre- and postconditions, but we do not need old expressions, as the state before and after are the same.

Both functions and methods can be *ghost* code. Ghost methods are also called *lemma* methods. Ghost code is code that is only used for verification, and is not present after the code is compiled. This means that we cannot execute these, but instead we can use the extended specification syntax in ghost functions and methods. Ghost functions are marked with `predicate!` in Prusti. Prusti currently does not support ghost methods.

Functions and methods can also be *trusted*. For a trusted function or method, the verification tool does not look at the body; it trusts that it is correct, and that the postconditions are satisfied at the end. Trusted ghost functions and methods do not have a body, we call them *abstract*.

In this thesis, we mostly use a pseudo-code-like syntax rather than actual Rust syntax with Prusti extensions. This is intended to be easier to read, and underscores that the approach we present is not specific to Prusti, but could also be implemented in other verification tools with similar features.

Chapter 3

Approach

3.1 Regular lock

To start, we describe how we add specifications to a lock. With a lock, multiple threads in the same process can temporarily obtain exclusive access to a shared resource. In Rust, `Mutex` implements a lock. When it is acquired, we obtain a `MutexGuard`, which can be used to access the protected resource.

3.1.1 Invariant

Even without specifications, the type system gives us some guarantees. When we create a lock, we have to provide a resource of the correct type. When we later acquire it, we obtain access to this resource, and we can be sure that it has the correct type. Finally, when releasing the lock, the resource must still have the correct type. If we try to violate these constraints, the compiler will raise a type checking error.

We extend this idea by adding a lock invariant. The lock invariant is a Boolean predicate over the type protected by the lock. We wrap the lock constructor in a method which takes the predicate as a parameter, in addition to the initial value. We also add a precondition which requires that the predicate is true for the given initial value. The lock `acquire` method gets a postcondition that the predicate is true for the value that is currently in the lock. And finally, `release` has a precondition that the predicate is still true for the possibly changed value. The verification tool will return an error if we violate the invariant.

Here is a summary of the lock API with specifications:

trusted method `new_lock`(*val*: `T`, *invariant*: `T` → `bool`): `Mutex<T>`
  requires *invariant*(*val*)

trusted method `acquire`(m: `&Mutex<T>`): `MutexGuard<T>`
  ensures $invariant(result.val)$

trusted method `release`(g: `MutexGuard<T>`)
  requires $invariant(g.val)$

3.1.2 Two-state invariant

The invariant restricts the value that the lock can have at any point in time, but sometimes we also want to restrict how the value changes over time. This is especially useful if the value changes monotonically in some dimension. As an example, consider a set protected by a lock, from which we never remove elements. If we know that the set contained a particular value in the past, we should be able to assert that this value is still in the set when we acquire the lock again.

To address this case, we add a two-state invariant to the lock, which is a predicate which takes two arguments of the type protected by the lock. $twostate(oldval, val)$ is true if the value in the lock can be val if it was $oldval$ at some point in the past. In the set example, we could define this as $twostate(oldval, val) := \forall x.\ x \in oldval \rightarrow x \in val$, or simply $oldval \subseteq val$.

The two-state invariant must be transitive. This ensures that it holds between any two points in time, even if we only check it between consecutive points. We enforce this by adding the definition of transitivity as a precondition

$$\forall a, b, c.\ twostate(a, b) \land twostate(b, c) \rightarrow twostate(a, c)$$

to the lock constructor. We also require reflexivity with $\forall v.\ twostate(v, v)$, so that it holds even if the value was not changed between some point in the past and now.

We check that the two-state invariant holds between the values at the time of `acquire`, and the matching `release`. We do this by remembering the value at the time of `acquire`, and then adding a precondition of the two-state predicate between the remembered old value and the current value to `release`.

Finally, we need a way to make use of the two-state invariant. For the regular invariant, this was simply a postcondition on `acquire`, but here it is more involved.

First, we introduce a Boolean predicate $oldvalue(v)$, which says that v was previously a value of the lock. This predicate has no body (it is 'abstract'), so initially we know nothing about its value. We add a postcondition to the lock constructor that $oldvalue$ is true for the initial value, and similarly to `release` for the value at that time.

Next, we introduce a lemma method `apply_old_value`($oldval$), which can be called while the lock is acquired. We can call it with an old value of the lock, and learn that the two-state invariant holds between that old value and the current value. This is implemented by having $oldvalue(oldval)$ as a precondition, and $twostate(oldval, val)$ as a postcondition, where val is the value at time of `acquire`.

To finish the set example, we can now use this to assert that values that were previously in the set are still in the set. Proof state annotations are marked with $\triangleright$, they show relevant facts as they are obtained. We could turn these annotations into assertions, and it would still verify.

let $m :=$ `new_lock`$(\{\},\ \lambda\ v.$ **true**$,\ \lambda\ oldv,\ v.\ oldv \subseteq v)$;
let $g :=$ `acquire`$(\&m)$;
$g.val := g.val \cup \{4\}$;
let $oldv := g.val$;
`release`(g);
$\triangleright\ oldvalue(oldv)$
let $g :=$ `acquire`$(\&m)$;
`apply_old_value`$(\&g,\ oldv)$;
$\triangleright\ twostate(oldv, g.val)$
assert $4 \in g.val$;
`release`(g);

Listing 3.1 shows the complete specification of the lock. Here we also add an identifier to the lock, which allows us to distinguish different instances. This way, *oldvalue* facts from one lock cannot be used on another lock. In the Rust implementation, we use `u32` as the `Id` type. Since we cannot actually refer to the *invariant* parameter of `new_lock` in the other methods, we 'copy' it into *m_invariant* by ensuring that they are equal for all v (same for *twostate*).

3.1.3 More on abstract functions

It is important to understand that when we learn that $oldvalue(v)$ is true for some v, *oldvalue* does not change (it is a mathematical function after all), only our knowledge about it changes. That means that $oldvalue(v)$ has always been true. Nevertheless, if we can assert $oldvalue(v)$, we can be sure that v was an old value in the past. In fact, if we could assert $oldvalue(v)$ when we only learn it in the future, that would allow circular reasoning.

One needs to be careful when creating trusted lemma methods. For example, if `apply_old_value`($oldval$) was instead defined with just a postcondition $oldvalue(oldval) \rightarrow twostate(oldval, val)$, which looks very similar to the actual definition, it would be unsound. This is because the implication does not force us to know that $oldvalue(oldval)$ is true right now. We can apply the implication later when we do learn $oldvalue(oldval)$, and can then arrive

struct `Mutex<T>` $\{id\colon \texttt{Id}\}$

struct `MutexGuard<T>` $\{id\colon \texttt{Id},\ \textbf{pub}\ val\colon \texttt{T},\ oldval\colon \texttt{T}\}$

abstract function $m\_invariant(id : \texttt{Id},\ v\colon \texttt{T})\colon \texttt{bool}$

abstract function $m\_twostate(id : \texttt{Id},\ oldv\colon \texttt{T},\ v\colon \texttt{T})\colon \texttt{bool}$

abstract function $m\_oldvalue(id : \texttt{Id},\ v\colon \texttt{T})\colon \texttt{bool}$

trusted method `new_lock`$(val\colon \texttt{T},\ invariant\colon \texttt{T} \to \texttt{bool},$
$twostate\colon \texttt{T} \times \texttt{T} \to \texttt{bool})\colon$ `Mutex<T>`
  requires $invariant(val)$
  requires $\forall v.\ twostate(v, v)$
  requires $\forall a, b, c.\ twostate(a, b) \wedge twostate(b, c) \to twostate(a, c)$
  ensures $\forall v.\ m\_invariant(result.id,\ v) = invariant(v)$
  ensures $\forall oldv, v.\ m\_twostate(result.id,\ oldv,\ v) = twostate(oldv,\ v)$
  ensures $m\_oldvalue(result.id,\ val)$

trusted method `acquire`$(m\colon$ `&Mutex<T>`$)\colon$ `MutexGuard<T>`
  ensures $m\_invariant(m.id,\ result.val)$
  ensures $result.id = m.id$
  ensures $result.val = result.oldval$

trusted method `release`$(g\colon$ `MutexGuard<T>`$)$
  requires $m\_invariant(g.id,\ g.val)$
  requires $m\_twostate(g.id,\ g.oldval,\ g.val)$
  ensures $m\_oldvalue(g.id,\ g.val)$

abstract method `apply_old_value`$(g\colon$ `&MutexGuard<T>`$,\ oldval\colon \texttt{T})$
  requires $m\_oldvalue(g.id,\ oldval)$
  ensures $m\_twostate(g.id,\ oldval,\ g.oldval)$

Listing 3.1: The complete lock specification.

at a contradiction, because $twostate(oldval, val)$ can be false for an $oldval$ which was not an old value at the time we called `apply_old_value`$(oldval)$.

3.1.4 Passing predicates to methods

The Prusti specification language is a first-order logic, which means that we cannot directly pass a predicate as an argument to a method. But how can we pass an invariant predicate of type $\texttt{T} \to \texttt{bool}$ to the lock constructor?

First, we define a `Predicate` type, which simply contains an `Id`. We define a `new_predicate` method, which returns a `Predicate` and has no pre- or postconditions. And we define an abstract, Boolean $predicate\_val(pred, val)$ function, which takes a `Predicate` and a value of a generic type.

To use it, we call `new_predicate` to obtain a `Predicate` *pred*, and then add an assume statement to define *predicate_val* for this *pred*. For example, the expression $val > 0$ is turned into the assume statement

$$\forall val.\ predicate\_val(pred, val) \leftrightarrow (val > 0).$$

We can now pass *pred* to a method, which can refer to the predicate through *predicate_val*. In the Rust implementation, we wrap this in a convenient macro which takes an expression and returns a `Predicate`.

3.1.5 Lock release in Rust

In Rust, when we acquire a `Mutex`, we obtain a `MutexGuard`. To release the lock, we simply 'drop' the `MutexGuard`. Dropping happens implicitly when a variable goes out of scope, and causes the destructor to be run, which in the case of `MutexGuard` releases the lock. This makes it convenient to work with and prevents forgetting to release.

However, because drop happens implicitly, this makes it hard to attach pre- or postconditions to it. Instead, we require users of our wrapped lock type to explicitly call a `release` method. To prevent users from circumventing this by just dropping the wrapped mutex guard, we wrap the `MutexGuard` in a `ManuallyDrop`. This means that when the guard is dropped, the lock remains locked. This ensures soundness at the cost of introducing a deadlock.

We see two ways to mitigate this: First, we can add a destructor to the guard which panics, which turns a deadlock into an easier to debug runtime panic. Second, a 'lint' could be introduced in the Rust compiler which can be attached to struct definitions resulting in a compile error when an instance is dropped, similar to the existing `must_use` and `must_not_suspend` lints. We could then attach this lint to the `MutexGuard` type.

3.2 Model

Our goal is to show refinement between a TLA$^+$ model and an implementation in Rust, using Prusti. To do this, we first need to translate the model from the TLA$^+$ language into Prusti constructs.

A basic TLA$^+$ model consists of variable declarations, and *Init* and *Next* operators. There may be a *Spec* operator which combines *Init* and *Next* into a single temporal formula, but we do not handle temporal formulas in our approach and require separate *Init* and *Next*. By convention, there usually is also a type invariant operator *TypeOK* (sometimes called *TypeInvariant*), which describes what types the variables have. An example is shown in Listing 3.2. The same example translated to Prusti is shown in Listing 3.3.

MODULE *counter_with_actions*

EXTENDS *Integers*, *Sequences*
VARIABLES *count*, *std_out*

$$TypeOK \triangleq count \in Int \land std\_out \in Seq(Int)$$

$$Init \triangleq count = 0 \land std\_out = \langle\rangle$$

$$Increment(v) \triangleq \begin{array}{l} \land\ v > 0 \\ \land\ count' = count + v \\ \land\ std\_out' = Append(std\_out, count) \end{array}$$

$$Print0 \triangleq \begin{array}{l} \land\ std\_out' = Append(std\_out, 0) \\ \land\ \text{UNCHANGED } count \end{array}$$

$$Next \triangleq \begin{array}{l} \lor\ \exists\, v \in Int : Increment(v) \\ \lor\ Print0 \end{array}$$

Listing 3.2: Example TLA$^+$ model.

In the example, there are two variables: *count*, and *std_out*. *TypeOK* tells us that *count* is an integer, while *std_out* is a sequence of integers. *Init* defines the initial state, and *Next* defines possible state transitions. *Next* is a disjunction of two cases: In a single step, either *count* is incremented by some positive integer, and the old value of *count* is appended to *std_out*, or 0 is appended while *count* remains unchanged.

We can translate the *Init* and *Next* TLA$^+$ operators into Prusti predicates. In TLA$^+$ operators, we can directly use variables, but in Prusti predicates, we need to pass them in a parameter. We translate all variables to fields in a `struct AbsState` to make this convenient. In *Next*, we use primed variables to refer to the next state, which we translate to a parameter `s` for the current state and `n` for the next state.

Next, we discuss how various features can be translated in more detail. In Section 3.2.5, we will look at how we could restrict TLA$^+$ to a subset which would allow to perform this translation mechanically.

3.2.1 Actions

The Prusti version of the example in Listing 3.3 has an `Action` type, and a parameter of this type in `next`, which does not exist in the TLA$^+$ version.

We expect *Next* to be written as a disjunction, where each arm may have existential quantifiers, and then applies an operator with the existentially quantified variables as parameters. This can then be translated into the `enum Action`, where each disjunction arm corresponds to an enum variant,

```
#[derive(Clone, Copy)]
struct AbsState {
  count: i32,
  stdout: AbsSeq<i32>,
}

#[derive(Clone, Copy)]
enum Action {
  Increment(i32),
  Print0,
}

predicate! {
  fn init(s: AbsState) -> bool {
    s.count == 0 &&
    s.stdout === AbsSeq::empty()
  }
}

predicate! {
  fn next(s: AbsState, n: AbsState, a: Action) -> bool {
    match a {
      Action::Increment(v) =>
        v > 0 &&
        n === AbsState {
          count: s.count + v,
          stdout: s.stdout.append(s.count),
          ..s
        },
      Action::Print0 =>
        n === AbsState {
          stdout: s.stdout.append(0),
          ..s
        },
    }
  }
}
```

Listing 3.3: The example model from Listing 3.2 translated to Prusti.

and the existential quantifiers correspond to fields on the variants. The `Action` is then passed in a parameter to `next`, which can be written as a `match` expression over the `Action`. We could write

```
exists(|a: Action| next(s, n, a))
```

to hide this parameter, which more closely matches *Next* in the TLA$^+$ version.

In the implementation, whenever a step in the model is taken, the `Action` that corresponds to this step must be provided. The main motivation for this is that automated verifiers can have difficulties proving disjunctions and existential quantifiers. By providing an `Action`, we can explicitly give witnesses for the quantifiers, avoiding this problem. Additionally, it communicates intent, and allows to easily find where in the implementation a particular action is performed.

3.2.2 Types

TLA$^+$ is an untyped language, where every value is a set. When an operator is used with a value of the wrong type (as in $2 \wedge \langle 5 \rangle$), the result is unspecified [4, p. 296]. Rust on the other hand is typed, and the type-checker prevents using values of the wrong type. In most places, the type is inferred, but we need to provide it in struct definitions, arguments and quantifiers.

While TLA$^+$ is untyped, there are sets that are used like types, such as *Int*, the set of all integers. In quantifiers, we can directly use these type sets, as in $\exists v \in Int$. And for variables, this information is provided in *TypeOK*.

Next, we show common TLA$^+$ type sets and corresponding Rust types.

BOOLEAN: Corresponds to `bool`.

Int: Prusti does not yet have unbounded integers, we can approximate this with bounded types such as `i32`.

SUBSET T: This is the powerset of T, or the type of sets of T. We have defined a corresponding type in Rust: `AbsSet<T>`.

$[S \to T]$: This is the set of functions $S \to T$. We represent this with a map type `AbsMap<S, T>`.

$Seq(T)$: Sequences of T, corresponds to `AbsSeq<T>`.

$[h_1 : S_1, \ldots, h_n : S_n]$: A record with n fields, where the i-th field has name h_i and is in S_i. This corresponds to a struct in Rust:

```
struct SomeStruct { h1: S1, ..., hn: Sn }
```

$S_1 \times \cdots \times S_n$: n-tuples where the i-th component is in S_i. This corresponds to the tuple in Rust: `(S1, ..., Sn)`.

$[type : \{\text{“t1”}\}, h1 : S1, \ldots, h2 : S2] \cup \cdots \cup [type : \{\text{“tn”}\}, h3 : S3, \ldots, h4 : S4]$: This is a tagged union type. TLA$^+$ has no builtin feature for this, so we build it as a union of record types, where *type* is the discriminant. Rust on the other hand does have enums built in:

```
enum SomeEnum {
  T1 { h1: S1, ..., h2: S2 },
  ...,
  Tn { h3: S3, ..., h4: S4 },
}
```

3.2.3 Expressions

In this section, we look at various TLA$^+$ expressions and operators, and how they can be translated to Prusti.

Logic

The Boolean operators $\wedge \vee \neg \Rightarrow \equiv$ correspond to `&& || ! ==> <==>`. Constants TRUE, FALSE correspond to `true`, `false`. Quantifiers $\forall x \in S : p$ and $\exists x \in S : p$ correspond to `forall(|x: S| p)` and `exists(|x: S| p)`.

Sets

Since all values in TLA$^+$ are sets, the set equality operators $=$ and $\neq$ correspond to `==` and `!=` for primitive types, and snapshot equality `===` and `!==` for all other types.

The set operators $x \in S$, $x \notin S$, $S \subseteq T$ are translated as `s.contains(x)`, `!s.contains(x)`, `s.is_subset(t)` on our `AbsSet<T>` type. The operators $\cup$, $\cap$, $\setminus$, UNION have not yet been implemented on `AbsSet<T>`, but it would be easy to do so: We can create a corresponding abstract function, and add its definition [4, p. 300] as a postcondition.

The set constructor $\{e_1, \ldots, e_n\}$ can be translated as

```
AbsSet::empty().add(e1) ... .add(en).
```

Set comprehensions $\{x \in S : p\}$ and $\{e : x \in S\}$ also have not yet been implemented, but could be, by first creating a function expression from p or e (discussed next), and then defining abstract functions which implement the set comprehensions, taking the function expression as a parameter, and with the definition of the set comprehensions in postconditions.

Functions

Remember that we represent functions with the `AbsMap<S, T>` type. Function application $f[e]$ corresponds to `f.get(e)`. The function expression $[x \in S \mapsto$

$e]$ can be translated to `AbsMap::<S, T>::new(e)` if e is independent of x. And the update expression $[f$ EXCEPT $![e_1] = e_2]$ is translated to `f.set(e1, e2)`.

Translating a general function expression $[x \in S \mapsto e]$ where e depends on x is a bit more involved. In Section 3.1.4, we showed how we can pass a predicate in a function parameter, however, this solution cannot be used in pure code. What we can do instead is to create a new abstract function `f` returning an `AbsMap<S, T>`, with postcondition `forall(|x: S| result.get(x) === e)`. Then we translate the function expression as `f()`. Any free variables in `e` need to be passed as parameters to `f`.

Records

The field access expression $e.h$ corresponds to `e.h` in Rust. The record constructor $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$ is translated to

```
SomeStruct { h1: e1, ..., hn: en }.
```

Record update $[r$ EXCEPT $!.h = e]$ can be written as

```
SomeStruct { h: e, ..r }.
```

Enums

Remember that enums are represented in TLA$^+$ as a union of record types, with an explicit discriminant field. The variant constructor $[type$: "`t1`", $h1$: $e1, \ldots, h2 : e2]$ is translated to

```
SomeEnum::T1 { h1: e1, ..., h2: e2}.
```

Field access is written $e.h$ in TLA$^+$, but in Rust, enum fields can only be accessed though pattern matching in a `match` or `if let` expression. This means we need to perform a more complex transformation of the TLA$^+$ code into Rust, where we introduce a `match` expression to replace discriminant tests such as $e.type =$ "`t1`", and replace the field accesses $e.h$ with fresh identifiers, which are bound in the patterns of the `match`.

Tuples

Tuple indexing $e[i]$ with $i \in \{1, \ldots, n\}$ for an n-tuple can be translated as `e.j`, where $j = i - 1$ (note that TLA$^+$ indices are 1-based, while Rust indices are 0-based). The tuple constructor $\langle e_1, \ldots, e_n \rangle$ is translated as `(e1, ..., en)`.

Other expressions

IF p THEN e_1 ELSE e_2 corresponds to `if p { e1 } else { e2 }`.

CASE $p_1 \to e_1 \square \cdots \square p_n \to e_n \square$ OTHER $\to e$ can be translated as an if-else chain `if p1 { e1 } ... else if pn { en } else { e }`. If there is

no OTHER part, we can end the if-else chain with `else { unreachable!() }`. However, this translation is only valid if the conditions p_i are mutually disjoint, and, when there is no OTHER, if always exactly one p_i is true.

This is because, in TLA$^+$, if multiple conditions are true, then it is unspecified which of the corresponding values is picked (it is implemented with CHOOSE), while the Rust version is guaranteed to pick the first value. Usually, this is not a problem, because picking the first value is a valid implementation of the TLA$^+$ spec. But we may be able to prove statements about the Rust version that do not hold for the TLA$^+$ version. There is also the edge case that in TLA$^+$, two CASE expressions that have the same set of possible values are guaranteed to yield the same value, which means we can prove e.g. $(\text{CASE TRUE} \to 1 \;\square\; \text{TRUE} \to 2) = (\text{CASE TRUE} \to 2 \;\square\; \text{TRUE} \to 1)$, which would be false in Rust. But not even TLC implements this edge case [4, p. 262]. To avoid all of this, the conditions of a CASE expression should be mutually disjoint [4, p. 298].

LET $d \triangleq f$ IN e can be translated to `{ let d = f; e }`.

We have not implemented the CHOOSE $x \in S : p$ expression, but it could be done in the same way as set comprehensions.

3.2.4 Constants

TLA$^+$ modules can contain constant declarations, e.g. CONSTANT C. These constants can be used in the model definition, and are intended for values that remain constant over time, but can differ depending on the situation in which the model is used. When checking a model with TLC, concrete values for these constants must be picked.

In addition, assumptions can be made about constants in ASSUME p statements, where p can refer to constants. The values that are picked for the constants must satisfy these assumptions. Theorems defined in the module can rely on them.

When translating this concept to Rust, we need to define constants in such a way that the `init` and `next` predicates can refer to them. We do it by declaring an abstract function without parameters for each constant. This means that references to constant C are replaced with `c()`.

To define the value of constant `c()`, a statement `prusti_assume!(c() === e)` for some expression `e` can be used. This should be inserted at the beginning of the program, `e` can for example depend on process arguments. And we can translate ASSUME p statements to `prusti_assert!(p)`, so that we check these assumptions. They should be placed after the `prusti_assume!` statements which assign the values of constants.

3.2.5 Mechanical translation

For the most part, the translation from TLA$^+$ to Prusti is straight forward. The translation of types, and particularly enums is more complex, because these are implemented very differently between the two. By restricting the TLA$^+$ code in certain ways, it would be possible to perform the translation automatically (though we have not implemented this). Here we describe the restrictions which would allow this automatic translation.

General

We require the TLA$^+$ module to have *Init*, *Next* and *TypeOK* operators.

Types

First, we define a 'type set' as one of BOOLEAN, Int, SUBSET T, $[S \to T]$, $Seq(T)$, $S_1 \times \cdots \times S_n$, or a name N, where: S, S_i, T are type sets. N is an operator, which is defined as either a type set (this corresponds to a type alias in Rust), or a struct or enum definition. A type set can be translated directly to the corresponding Rust type.

The *TypeOK* operator must be a conjunction of $x \in T$ expressions, one for each variable x, where T is a type set. Each quantifier must be of the form $\forall\, x \in T : p$ or $\exists\, x \in T : p$, where T is a type set. The field types in struct and enum definitions must be type sets. These requirements ensure that we have a type set in all places where the corresponding Rust syntax requires a type.

Enum access

If e is an expression of enum type, with variant names `t1`, ..., `tn`, then $e.type$ may only appear as part of an expression IF $e.type =$ "`ti`" THEN e_i ELSE f or $e.type =$ "`ti`" $\wedge\, e_i$ or CASE $e.type =$ "`t1`" $\to e_1 \;\square\; \cdots \;\square\; e.type =$ "`tn`" $\to e_n$. Field accesses $e.h$ may only appear inside e_i in the expressions above, where h must be a field of variant `ti`. This ensures that enum access can be translated to a `match` expression.

Type hints

When constructing a struct or enum instance in Rust, we need to give the name of the struct or enum definition. This information is not present in TLA$^+$, but we can add it as follows: We define an operator $Type(v, T) \triangleq v$, and require all struct or enum constructor or update expressions v to be wrapped as $Type(v, T)$, where T is the name of the struct or enum type definition.

We can also uses this as a type hint in places where a type cannot be inferred. For example, `AbsSet::<T>::empty()` has a type parameter which can be

inferred and thus omitted in many cases, but not always. We can then provide the type as $Type(\{\}, \text{SUBSET } T)$.

Well-typedness

We require all expressions to be well-typed. For example, in the expression IF p THEN e_1 ELSE e_2, e_1 and e_2 must have the same type, and p must have type BOOLEAN. We can rely on the type checker of the Rust compiler to check this.

3.3 Ghost lock

Now that we have translated the TLA$^+$ model into Rust, the next step is to connect the model to the implementation. The model state is an abstraction of the state of the program and its environment, and the *next* relation defines the possible state changes in this system. Our goal is to verify that the program refines the model, which means that whenever it takes a concrete step which changes the abstract view of the system, the corresponding abstract step is permitted by the model.

But how do we know which abstract step corresponds to a given concrete step? The idea of the approach is that the program just tells us. We embed the entire abstract state into the program as ghost variables, and whenever the program takes a concrete step which is visible in the abstract view, it also changes the ghost variables accordingly. Except, the environment part of the abstract state cannot be changed by the program directly, but only by calling the methods which interact with the environment. This way, the observable behavior of the program matches the model, while the internal implementation is not constrained.

The model may change multiple variables in a single atomic step, so we need a way to group multiple ghost variable updates together. This is where the ghost lock comes in. We can acquire and release the ghost lock, and update the ghost variables between the two points, and all these updates must correspond to a single step of the model.

Like the regular lock, the ghost lock is implemented as a Rust struct, with various fields and methods. But unlike the regular lock, it is ghost state, which means that it only exists for specification and verification purposes, and is not present at runtime. This means that the state in the ghost lock cannot affect the execution of the program; for example, we cannot print a value stored in the model state.

The abstract state represents a global view of the system. This system can consist of multiple threads, or even distributed processes communicating over a network, all making steps in this single global model of the system.

```
struct AbsState {count: i32}

enum Action = Increment(i32)

ghost function init(s: AbsState): bool
  s.count = 0

ghost function next(s: AbsState, n: AbsState, a: Action): bool
  match a {
    Action::Increment(v) ↦
      v > 0 ∧ n = AbsState{count : s.count + v}
  }

method main(){
  let mut gl := GhostLock::new(AbsState{count: 0});
  gl.acquire();
  gl.state.count := gl.state.count + 4;
  gl.release(Action::Increment(4));
}
```

Listing 3.4: A simple model is defined, which allows a field to be incremented by a positive amount. The `main` method creates a ghost lock instance and then performs a ghost lock step in which the field is incremented.

Some steps of the model may not even be performed by software, e.g., a button being pressed by a user.

Next, we describe how the ghost lock is used and implemented in more detail.

3.3.1 Initializing

Before the ghost lock can be used, an instance of it must be created. This is done with the method `GhostLock::new(`*init_state*`: AbsState): GhostLock`. It should be called at the start of the program, and takes the initial state as a parameter. We check that it is a valid initial state according to the model, with a precondition *init*(*init_state*).

If the system consists of multiple threads or even distributed processes, then each thread needs to create its own instance of the ghost lock. The user of the approach must ensure that only one instance is created in each thread, and that each instance is created with the same initial state.

3.3.2 Acquiring and releasing

Listing 3.4 shows a simple example of how the ghost lock is used. We can acquire the ghost lock with the `acquire` method, read and write the model state through the *state* field, and finally release it. We call this a *ghost lock*

step. The *state* field can be freely modified, but we can only release if the modifications form a valid step of the model.

We add a field *locked*: `bool` to the `GhostLock` to keep track of whether the ghost lock is currently acquired. `acquire` has a precondition $\neg self.locked$ and postcondition $self.locked$, and conversely `release` has a precondition $self.locked$ and postcondition $\neg self.locked$. This ensures that release and acquire calls are matched. The *locked* field and all other fields introduced later are private, to protect them from changes outside the ghost lock implementation. Only the *state* field is public. The private fields can still be read, through getter functions with the same name as the fields, e.g. $gl.locked()$.

The `release` method takes an `Action` as a parameter, which describes the step in the model that was taken between acquire and release. We check that this step was actually taken with a precondition

$$next(self.initial\_state,\ self.state,\ action).$$

The *initial_state* field of the `GhostLock` refers to the value of $self.state$ at time of the call to `acquire`. This is done with a postcondition $self.initial\_state = self.state$ on `acquire`.

There is also a `release_stutter` method, which requires $self.initial\_state = self.state$. This can be used to perform a stutter step, which does not change the state; this is useful if we only need to read the state.

3.3.3 Interacting with the environment

So far, we can use the ghost lock to take steps in the model state, and it is verified that these are valid steps. But nothing connects this to what the program is actually doing yet. In our approach, we do not directly link model variables to internal program variables. Instead, we only care about how the program interacts with its environment, and check that this corresponds to the model.

To do this, we declare some model variables to be *environment variables*. We add specifications to all methods which interact with the environment, to describe how this interaction is reflected on the environment variables. We also disallow changing these variables directly; they may only be changed by calling the methods which actually interact with the environment. An example of this is shown in Listing 3.5.

To protect the environment variables from change, we create a copy of `AbsState` named `EnvState`, which contains only the environment variables. We also define a function *get_env_state*, which projects an `AbsState` to an `EnvState`. Then, we add the field *env_state*: `EnvState` to the `GhostLock`. We use this to keep a protected copy of the environment variables in

```
struct AbsState {count: i32, stdout: AbsSeq<i32>}

enum Action = Increment(i32) | Print0

ghost function init(s: AbsState): bool
  s.count = 0 ∧ s.stdout = AbsSeq::empty()

ghost function next(s: AbsState, n: AbsState, a: Action): bool
  match a {
    Action::Increment(v) ↦ v > 0 ∧
      n = AbsState{count: s.count + v, stdout: s.stdout.append(s.count)}
    Action::Print0 ↦ n = AbsState{stdout: s.stdout.append(0), ..s}
  }

struct EnvState {stdout: AbsSeq<i32>}

function get_env_state(s: AbsState): EnvState
  EnvState{stdout: s.stdout}

trusted method print(x: i32, gl: &mut GhostLock)
  requires gl.locked()
  ensures gl.locked()
  requires gl.env_state() = get_env_state(gl.state)
  ensures gl.env_state() = get_env_state(gl.state)
  ensures gl.initial_state() = old(gl.initial_state())
  ensures gl.state = AbsState{
    stdout: old(gl.state.stdout).append(x),
    ..old(gl.state)
  }

method main(){
  let mut gl :=
    GhostLock::new(AbsState{count: 0, stdout: AbsSeq::empty()});
  let mut c := 0;
  while c < 5 {
    invariant ¬gl.locked();
    gl.acquire();
    assume c = gl.state.count;
    print(c, &mut gl);
    c := c + 4;
    gl.state.count := gl.state.count + 4;
    gl.release(Action::Increment(4));
  }
}
```

Listing 3.5: The counter example is extended by printing the value of the counter. The `print` method interacts with the environment, which is modeled with the *stdout* environment variable. *stdout* is part of `EnvState` in order to protect it from changes made directly without calling `print`.

the *state* field. To ensure that they stay in sync, we add a postcondition $self.env\_state = get\_env\_state(self.state)$ to `acquire`, and the same as a precondition to `release`. The result is that, if we accidentally modify an environment variable through *gl.state*, verification will fail.

For the methods which interact with the environment, we add a parameter taking a mutable reference to the ghost lock, and we add trusted specifications. We need to check in preconditions that the ghost lock is locked and the *env_state* is valid. In a postcondition, we describe how the state is updated, and another postcondition states that the *env_state* is also updated accordingly. We also need to provide postconditions for the things that have not changed: The ghost lock is still locked, and *initial_state* still has the same value.

In the example in Listing 3.5, we also show how this is used in the `main` method. We have a local variable *c* which corresponds to the model variable *count*, and we print and increment it in a loop. After acquiring the ghost lock, we initially know nothing about the abstract state, so we need to assume that *c* is equal to *count*. We will see later in Section 3.4 how we can remove this assumption.

3.3.4 Reasoning with preserved predicates

With the regular lock, we were able use the `apply_old_value` lemma method to learn something about the current value from a past value and the two-state invariant (see Section 3.1.2). We can do something similar with the ghost lock.

An example is shown in Listing 3.6. In the example, we can send and receive messages over a network. The network is modeled as a set of messages that have been sent, and when we receive a message, we learn that this message is in the set of sent messages. Concretely, one side can send a ping message, and if the other side has received the ping, it can reply with a pong. The `main` method first receives a message, and if it is a ping, it responds with a pong.

The receive and send are in two separate ghost lock steps. In the first step, we learn that *msg* is in *msgs*, and in the second step, we need this knowledge, as it is a precondition of the `SendPong` action. Again, we initially know nothing about the state after acquiring the ghost lock, so we need some way to transfer this knowledge from the first to the second step, taking advantage of the fact that messages are never removed from *msgs*.

This is why we have the call to the `preserved_predicate` method. It takes an old state of the ghost lock, and a predicate, which must hold for the old state and must be preserved by every possible step. Then we can conclude that it still holds in the current state.

```
enum Message = Ping(i32) | Pong(i32)

struct AbsState {msgs: AbsSet<Message>}

struct EnvState {msgs: AbsSet<Message>}

enum Action = SendPing(i32) | SendPong(i32)

ghost function init(s: AbsState): bool
  s.msgs = AbsSet::empty()

ghost function next(s: AbsState, n: AbsState, a: Action): bool
  match a {
    Action::SendPing(v) ↦
      n = AbsState{msgs: s.msgs.add(Message::Ping(v))}
    Action::SendPong(v) ↦ s.msgs.contains(Message::Ping(v)) ∧
      n = AbsState{msgs: s.msgs.add(Message::Pong(v))}
  }

trusted method send(msg: Message, gl: &mut GhostLock)
  (pre- and postconditions for locked and env_state omitted)
  ensures gl.initial_state() = old(gl.initial_state())
  ensures gl.state = AbsState{
    msgs: old(gl.state.msgs).add(msg),
    ..old(gl.state)
  }

trusted method receive(gl: &GhostLock): Message
  (preconditions for locked and env_state omitted)
  ensures gl.state.msgs.contains(result)

method main(){
  let mut gl := GhostLock::new(AbsState{msgs: AbsSet::empty()});
  gl.acquire();
  let msg := receive(&gl);
  let receive_state := gl.state;
  gl.release_stutter();
  if let Message::Ping(v) := msg {
    gl.acquire();
    gl.preserved_predicate(receive_state, λ s. s.msgs.contains(msg));
    send(Message::Pong(v), &mut gl);
    gl.release(Action::SendPong(v));
  }
}
```

Listing 3.6: Verified implementation of a Pong server. In order to use facts learned during previous ghost lock steps, it uses the `preserved_predicate` method.

We will now show how this is implemented. First, we need to keep track of old values of the ghost lock, and we do it the same as for regular locks, by introducing an abstract function $old\_state(s\colon \texttt{AbsState})\colon \texttt{bool}$. We then add postcondition $old\_state(\textbf{old}(self.state))$ to `release` and `release_stutter`, and $old\_state(init\_state)$ to `new`. This means that if we can assert $old\_state(v)$ for some v, then we know that v is a previous value of the ghost lock at that point.

For the `preserved_predicate` method, we check in preconditions that the ghost lock is locked, that the given old state is actually an old state using old_state, and that the given predicate holds for the old state. We also check that the predicate is preserved by all possible steps, with

$$\forall s, n, a.\ predicate(s) \wedge next(s, n, a) \rightarrow predicate(n).$$

If all these conditions hold, then we conclude in a postcondition that the predicate holds for the current state.

3.3.5 Linearizability

The model describes a set of behaviors which consist of a sequence of atomic steps. For a program to correctly refine the model, each ghost lock step must be atomic with respect to other steps, and we must be able to totally order all these steps, together with the steps taken by the environment, such that we obtain a valid sequential behavior of the model. In other words, the ghost lock steps must be *linearizable*.

Consider the example is Listing 3.7. The model allows to print 1 and 2 in a single step. Now imagine that we have two threads executing the `bad` method concurrently. It is possible that the `print` calls are interleaved, such that we get the output "1, 1, 2, 2". This behavior is not allowed by the model. The `bad` method does not refine the model, because the ghost lock steps are not linearizable. The `good` method on the other hand acquires a regular lock on the standard output before writing 1 and 2. No matter how many threads we run, we will never get a behavior which is not allowed by the model, because the ghost lock steps are linearizable. We will see later why the regular lock `acquire` and `release` methods take a reference to the ghost lock.

How can we systematically determine whether a ghost lock step is linearizable? We use the reduction argument by Lipton [5] for this, which we introduced in Section 2.2. This means that we can treat a ghost lock step as atomic if it consists of a sequence of right movers, followed by an arbitrary atomic action and a sequence of left movers. The idea is that we can move all actions to a single point in the middle, which is the linearization point. Ghost lock steps are then ordered according to the linearization points.

Right movers are actions which can be moved 'to the right' (i.e. later in time) across actions in other threads, without changing the outcome. Essentially,

```
struct AbsState {stdout: AbsSeq<i32>}

enum Action = PrintOneTwo

ghost function next(s: AbsState, n: AbsState, a: Action): bool
  match a {
    Action::PrintOneTwo ↦
      n = AbsState{stdout: s.stdout.append(1).append(2)}
  }

method bad(gl: GhostLock)
  requires ¬gl.locked()
  ensures ¬gl.locked()
{
  gl.acquire();
  print(1, &mut gl);
  print(2, &mut gl);
  gl.release(Action::PrintOneTwo);
}

method good(gl: GhostLock)
  requires ¬gl.locked()
  ensures ¬gl.locked()
{
  gl.acquire();
  let ol = stdout.acquire(&mut gl);
  ol.write(1, &mut gl);
  ol.write(2, &mut gl);
  ol.release(&mut gl);
  gl.release(Action::PrintOneTwo);
}
```

Listing 3.7: This example demonstrates linearizability. The `bad` method is not linearizable, because with two threads running it concurrently, the `print` calls could be interleaved. The `good` method on the other hand is linearizable.

right movers are actions that acquire a resource, for example receiving on a stream (e.g. TCP socket, standard input) to which we have exclusive access, or acquiring a regular lock which protects an environment resource. `preserved_predicate` is also a right mover because it 'acquires' an *old_state* fact, which could come from another thread. Left movers are the dual of right movers, examples are sending on a stream, or releasing a regular lock.

We call actions that are neither right nor left movers *non-movers*. There can be at most one such action per ghost lock step. Examples for non-movers are reads and writes to shared state where updates are immediately visible, such as files, GPIO pins, or atomic variables shared between threads.

Actions that are both right and left movers are *both-movers*; these are actions that are only visible in the current thread, such as reads and writes to local variables or the ghost lock state. There are no restrictions for both-movers.

There are also methods like remote procedure calls (e.g. HTTP request), which consist of a send and receive, and need to be represented by two separate ghost lock steps. But if we assume that the RPC is handled atomically at the other end, then we can simplify it to a single non-mover action.

We can now apply these rules to the example in Listing 3.7. `print` is a non-mover, so `bad` is not allowed. Internally, `print` is implemented as acquiring the lock on stdout, writing the argument and releasing the lock. In `good` on the other hand, we have a right mover (`acquire`), followed by three left movers (`write` and `release`), which means that this can be treated as an atomic step.

Verifying linearizability

The question is now, how can we verify this? Our idea is to add a Boolean flag, *did_act*, to the `GhostLock`, which represents whether we are past the linearization point. After acquiring the ghost lock, the flag is initially false, this is provided by a postcondition $\neg self.did\_act$ on `acquire`. For a right mover method, we add a precondition that the flag is false, and ensure that it is still false in a postcondition. For a non-mover, we also add a precondition that the flag is false, but in the postcondition the flag is true. Finally, a left mover has no added precondition, but a postcondition that the flag is true. With this setup, we can ensure that after a non-mover or left mover is called, only left movers can be called until the ghost lock is released. In fact, we can drop the postconditions that the flag is true from non-movers and left movers, because we do not actually need to know that the flag is true, it is sufficient that we know nothing about its value.

We can now see why the `acquire` and `release` methods of the stdout lock take a reference to the ghost lock: This is needed to mark them as right and

```
method bad(){
  let mut g1 = ghostlock_acquire();
  let mut g2 = ghostlock_acquire();
  let ol = stdout.acquire();
  ol.write(1, &mut g1);
  ol.write(1, &mut g2);
  ol.write(2, &mut g1);
  ol.write(2, &mut g2);
  ol.release();
  g1.release(Action::PrintOneTwo);
  g2.release(Action::PrintOneTwo);
}
method good(){
  let mut g1 = ghostlock_acquire();
  let mut g2 = ghostlock_acquire();
  let ol = stdout.acquire();
  ol.ghost_acquire(&mut g1);
  ol.write(1, &mut g1);
  ol.write(2, &mut g1);
  ol.ghost_release(&mut g1);
  ol.ghost_acquire(&mut g2);
  ol.write(1, &mut g2);
  ol.write(2, &mut g2);
  ol.ghost_release(&mut g2);
  ol.release();
  g1.release(Action::PrintOneTwo);
  g2.release(Action::PrintOneTwo);
}
```

Listing 3.8: Example with hypothetical interface allowing reentrancy, using the same model as Listing 3.7.

left movers respectively. We also allow these to be called while the ghost lock is not locked, because they do not interact with the ghost lock state.

3.3.6 Reentrancy

A design choice we had to make was whether the ghost lock should be reentrant. Reentrancy means that it is possible to perform a ghost lock step, while another step is currently ongoing in the same thread. We can either allow reetrancy, then we need to ensure that it does not lead to unsoundness. Or we can disallow it, and then we need to check that it does not occur.

Allow Reentrancy by itself does not seem to be a problem, it is not that much different from two ghost lock steps happening concurrently in different threads. We just need to ensure that the left and right movers of one step can move across the actions of the other step in the same thread.

Consider the method `bad` in Listing 3.8. Here we use a hypothetical interface to acquire two ghost lock guards, and then write to stdout after acquiring the regular lock on it. Obviously, this should not be allowed, because the output will be interleaved in a way not permitted by the model. The problem is that the lock on stdout is only exclusive to the current thread, when we need something stronger: We need the lock exclusively for one ghost lock guard. In method `good`, we show how this could be solved: After acquiring the regular lock, we also call a ghost method to lock it exclusively to one ghost lock guard. We do this separately to allow a regular lock to remain locked across multiple ghost lock steps.

Disallow If we disallow reentrancy, we need to ensure that it is not possible. A `GhostLock` instance cannot be acquired again if it is already locked. The user needs to ensure that at most one instance is created in each thread. Additionally, we make the `GhostLock` type `!Send` in the Rust implementation, which prevents transferring an instance to another thread, where we would then have two instances.

Comparison When we disallow reentrancy, we need to pass the mutable reference to the ghost lock through parameters to all methods which use it. We could avoid this when we allow it. However, this would have the disadvantage that it becomes easier to omit a ghost lock release call, since we are not forced to return the ghost lock in a released state. To address this, we could have a single `GhostLock` with a counter tracking how many ghost lock guards are currently open.

There could be cases where reentrancy is useful, such as when we call some method while in a ghost lock step, and that method needs to perform a different ghost lock step.

It is not yet clear which option is better. For this thesis, we have decided to disallow reentrancy.

3.4 Guards

In Listing 3.5, we needed to assume that the local variable c is equal to the model variable *count* after acquiring the ghost lock. In this section, we will see how we can use guards to get rid of this assumption.

The reason we needed the assumption is that after acquiring the ghost lock, we initially know nothing about the abstract state. This is because between

the release and acquire, other ghost lock steps could happen, for example in another thread. In this case, there are no other threads, so we know that the assumption is correct. But we want to avoid making assumptions, because it is very easy introduce unsoundness with incorrect assumptions. We want to say that only *this* thread is allowed to perform the `Increment` action, and then conclude that $c = gl.state.count$, instead of assuming it.

A *guard* is a ghost resource. Importantly, each guard must only exist once in the system; there cannot be two identical guards.

After defining the guards, we then disallow certain transitions of the model without ownership of a particular guard. For example, we can define a guard `CountGuard`, and only allow changing *count* if the thread owns the `CountGuard`.

Next, we can define a predicate over the model state and prove that its value cannot be changed without access to the guard. This now allows us to keep track of the value of these predicates, while not holding the ghost lock. Because the thread owns the guard, and the guard is unique, no transitions which require the guard can happen elsewhere.

In the original approach [1], guards were used in some of the case studies. While the idea is the same, the way we implement it in our approach is quite different. In Listing 3.9, we show how we can replace the assume statement with guards in the example. Next, we will explain how guards are used and how they are implemented.

3.4.1 Extending the model

Guard kinds As a first step, we define which guards can exist, with the `GuardKind` enum. Variants of this enum can have fields, such as thread or process identifiers. Usually, there is one guard for each agent in the system which has some local state. In the example, we have only one guard kind.

Requiring guards for actions Next, we define the *action_requires_guard* function, which defines which guards are required for which action. The value of the function is true if the guard of kind k is required for action a. In the example, the `Increment` action requires the `Counter` guard, while the `Print0` action does not require guards. If there are multiple agents, then the action usually has a parameter identifying the agent performing the action. We can then require the guard with the matching agent identifier.

3.4.2 Creating a guard instance

Guards are represented by the `Guard` struct. We can create an instance with the `Guard::new` method, which takes the guard kind as a parameter. The

```
enum GuardKind = Counter

ghost function action_requires_guard(a: Action, k: GuardKind): bool
  match a {
    Action::Increment(v) ↦ k = GuardKind::Counter
    Action::Print0 ↦ false
  }

method main(){
  let init_state := AbsState{count: 0, stdout: AbsSeq::empty()}
  let mut gl := GhostLock::new(init_state);
  let mut guard := Guard::new(GuardKind::Counter);
  let mut guard_state := init_state;
  let mut c := 0;
  while c < 5 {
    invariant ¬gl.locked();
    invariant guard.kind() = GuardKind::Counter;
    invariant released(guard.last_id());
    invariant final_state(guard.last_id()) = guard_state;
    invariant guard_state.count = c;
    gl.acquire();
    guard.open(&gl);
    guarded_preserved_predicate(
      guard_state, gl.state, guard.kind(), λs. s.count = c);
    print(c, &mut gl);
    c := c + 4;
    gl.state.count := gl.state.count + 4;
    guard_state := gl.state;
    gl.release(Action::Increment(4));
  }
}
```

Listing 3.9: Example with guards. The model is the same as in Listing 3.5. We protect the `Increment` action with the `Counter` guard. After opening the guard, we can then use `guarded_preserved_predicate` to learn that the value of *count* is still the same as at the end of the last step where we opened the guard.

user has to ensure that a guard of each kind is only created once in the entire system.

The guard kind is stored in the *kind* field. All fields of the guard are private and can be read with getter functions.

3.4.3 Opening a guard

When we want to use a guard in a ghost lock step, we call the `open` method. After opening the guard, we are then allowed to perform an action which requires the guard.

The ghost lock `release` method will need to check whether the required guards have been opened, so we need some way to remember which guards have been opened in the current ghost lock step. To do this, we add a field *id*: `Id` to the ghost lock, which identifies the current step of the ghost lock. The `acquire` method has no postcondition about the *id* field, so we can think of it as assigning a new id each time the ghost lock is acquired. Then, we create an abstract function $guard\_is\_open(id\colon \texttt{Id},\ kind\colon \texttt{GuardKind})\colon \texttt{bool}$, which means that the guard of this kind has been opened in the ghost lock step with this id. `open` gets a postcondition $guard\_is\_open(gl.id,\ self.kind)$, so we learn this fact when we open a guard. Finally, we can now check whether required guards were opened by adding to `release` the precondition

$$\forall k.\ action\_requires\_guard(action,\ k) \rightarrow guard\_is\_open(self.id,\ k).$$

Because we now have the *id* field on the ghost lock, we need to add the postcondition $gl.id() = \mathbf{old}(gl.id())$ to environment methods, so that the value is preserved across them. Previously, we had such a postcondition for the *initial_state* field. But we can now also attach this to the *id*: We introduce an abstract function $initial\_state(id\colon \texttt{Id})\colon \texttt{AbsState}$, and replace all uses of $gl.initial\_state()$ with $initial\_state(gl.id())$. The *initial_state* field is no longer needed, and we can remove the postcondition preserving it from the environment methods.

3.4.4 Reasoning with guard-preserved predicates

The last remaining step before we can remove the assume statement in the example is to take advantage of the fact that actions protected by a guard that we own cannot happen elsewhere. We do this similarly to the preserved predicates we saw in Section 3.3.4, where we passed a predicate that had to be preserved by any possible steps. But this time, the predicate only has to be preserved by steps which do not require the guard that we hold.

Because the guard is unique, we know that when we open a guard, no ghost lock steps requiring the guard have happened between the last step

where the guard was opened, and the current step. To be able to express this formally, we need some way to refer to the final state of the last step where the guard was opened. If we tried to store this state in a field of the guard, it seems tricky, because we only know the final state when `release` is called, where we do not have access to the guard instance. Instead, we use an indirection through the *id* field of the ghost lock. We add a field *last_id*: `Id` to the guard, which is set to the current ghost lock step id in `open` by a postcondition $self.last\_id = gl.id$. Then we introduce the abstract function $final\_state(id\colon \texttt{Id})\colon \texttt{AbsState}$, analogous to *initial_state*. We add a postcondition $final\_state(\mathbf{old}(self.id)) = \mathbf{old}(self.state)$ to both `release` and `release_stutter`. Now, we can refer to the final state of the last step where the guard was opened with $final\_state(self.last\_id)$.

What if we open a guard for the first time? When we create a guard, the *last_id* field is set to the constant id $INIT\_ID$, representing the initialization of the ghost lock. The ghost lock constructor has a postcondition $final\_state(INIT\_ID) = init\_state$. That way, the first time the guard is opened, the last state is actually the initial state of the ghost lock.

The actual reasoning is done in the separate `guarded_preserved_predicate` method. We could also have added this to `open` instead, but this way allows to have multiple preserved predicates in the same ghost lock step. The information of the last state where the guard was used needs to be passed from `open` to `guarded_preserved_predicate`, which we do with an abstract function $guarded\_twostate(s\colon \texttt{AbsState}, n\colon \texttt{AbsState}, k\colon \texttt{GuardKind})\colon \texttt{bool}$. The meaning of this is that there exists a (possibly empty) sequence of steps to get from state s to n without using guard k. `open` has a postcondition $guarded\_twostate(final\_state(\mathbf{old}(self.last\_id)), initial\_state(gl.id), self.kind)$.

`guarded_preserved_predicate` takes four arguments: *last_state*: `AbsState`, *state*: `AbsState`, *kind*: `GuardKind` and *predicate*: $\texttt{AbsState} \to \texttt{bool}$. The first three arguments match the arguments of *guarded_twostate*, and we have a precondition $guarded\_twostate(last\_state, state, kind)$. It works similarly to `predicate_preserved`: The predicate must hold for the old state and be preserved by all actions which do not require the guard, and then we learn that the predicate holds in the current state. To check that the predicate is preserved, we have the precondition

$$\forall s, n, a.\ predicate(s) \land next(s, n, a) \land \neg action\_requires\_guard(a, kind) \to predicate(n).$$

In the example in Listing 3.9, we store a copy of the state where the guard was last used in the variable *guard_state*. This is then used as the first argument to `guarded_preserved_predicate`. The *count* field can only be changed with the `Increment` action, which is protected by the guard. This means that the

predicate $\lambda s.\ s.count = c$ is preserved without access to the guard. We know that $guard\_state.count = c$ and learn that $gl.state.count = c$, which is exactly what we need to replace the assume statement from Listing 3.5.

3.4.5 Ensuring soundness

We said that there exists a possibly empty sequence of steps to get from the last state to the current state without using the guard. But this is only true if the linearization point of the last ghost lock step is before the one of the current step. A guard can be sent between two threads, even while there is an ongoing ghost lock step in both threads, so we need to ensure that this is sound.

The `open` method is a right mover, which means that the linearization point of the current step is after the call to `open`. We also need to ensure that the linearization point of the last step is before the call. Actually, we strengthen this to the requirement that the last step has been released. To implement it, we introduce the abstract function $released(id\colon \texttt{Id})\colon \texttt{bool}$, which means that the ghost lock step with this id has been released. We add the postcondition $released(\textbf{old}(self.id))$ to `release` and `release_stutter`, and $released(INIT\_ID)$ to `GhostLock::new`. Then, we can check the requirement by adding precondition $released(self.last\_id)$ to `open`.

3.4.6 Opening guards read-only

Guards can be sent between threads, for example using a `Mutex`. Rust also has a `RwLock`, which allows a single thread to obtain write access, or multiple threads to obtain read-only access at a time. We would like to be able to put a guard into a reader-writer lock, and allow multiple readers to open the guard concurrently in a read-only mode, allowing them to conclude that the guard has not been used since the last time the guard was opened writably, but not permitting to perform actions which require the guard. We have not implemented this in our code, but we show our ideas here.

To implement this, we add a new method `open_readonly`, which takes $\&self$ instead of **&mut** $self$. It has the same preconditions as `open`, i.e., $gl.locked$, $\neg gl.did\_act$, and $released(self.last\_id)$. But we only have the single postcondition $guarded\_twostate(final\_state(self.last\_id),\ initial\_state(gl.id),\ self.kind)$. This means that the $last\_id$ is not updated, and we do not learn $guard\_is\_open$, so we cannot perform actions which require the guard, but we can use `guarded_preserved_predicate`.

The challenge, however, is to ensure soundness. We still require that the last ghost lock step where the guard was used has been released; this part works the same way as before. But we still need to ensure that guard is not opened writably in other threads before the linearization point of the

current ghost lock step. Since we cannot change *last_id*, the other thread is not prevented from opening the guard. We need some other way to ensure that the read-only reference to the guard lives until after the linearization point.

Solution 1 One way to achieve this is with the lifetime system of Rust. We can specify lifetime bounds, which require one lifetime to live at least as long as another lifetime. In this case, we want the lifetime `'a` of the `&'a Guard` to live at least as long as the region between acquire and release of the ghost lock. To do that, we need a lifetime for this region. We can return a struct containing `PhantomData<Cell<&'b ()>>` from `acquire`, which has to be given back at `release`; the lifetime `'b` is now exactly the region between acquire and release. The downside of this solution is that we have this extra struct which has to be passed back to acquire in every ghost lock step, even when we do not use read-only guards.

Solution 2 Alternatively, we could add a new field to the ghost lock, to keep track of which guards have been opened read-only, and with which *last_id*. Then, we add another method to 'close' the guard, which is a left mover and thus after the linearization point. This method checks that the *last_id* is still the same, and then removes the guard from the set of opened read-only guards. In `release`, we check that this set is empty. With this solution, we ensure that each read-only open is matched by a close after the linearization point, and that the guard has not been used in the mean-time. This solution is also somewhat inconvenient, since it requires two calls for each read-only opened guard, and needs an additional field on the ghost lock which needs to be preserved by all environment functions.

Solution 3 Instead, we can also attack the problem from another angle: We can require all methods that allow sending values to another thread to be made ghost lock aware, only allowing sending while the ghost lock is not locked, or as a left mover. This is enough to ensure that the guard is not used before the linearization point, since it can only be sent to other threads after the linearization point. For completeness, the receiving side should similarly be outside a ghost lock step, or a right mover. This looks like the best solution, because it requires little additional code (passing the ghost lock to methods that send/receive values), and these restrictions are also useful elsewhere. For example, for locks which protect environment resources, we need the mover restrictions anyway.

There is a limited number of ways to send/receive values: Locks, channels, `Arc`, and thread spawn/join. `Arc` is the atomic reference-counting pointer. When an `Arc` is dropped and only one reference remains, this reference can use `get_mut` to obtain write access to the value. This means that the drop

implementation of `Arc` can be used to send values [7]. Since we cannot attach specifications to drop, we need to replace it with a regular method call and prevent dropping, as we did for `MutexGuard` in Section 3.1.5.

3.5 Atomics

In this section, we will see how we can verify programs that use atomics using our approach. *Atomics* are variables in memory that can be read and written concurrently by multiple threads, allowing communication between them. Each access is performed atomically. There are also more complex operations available, such as *compare and exchange*, which atomically compares the value with a first argument, and if it matches, sets it to a second argument.

For each access, we must choose an *ordering*, which restricts to what extent the compiler and processor can reorder accesses. Rust uses the C++ ordering model. There are three different orderings: *Relaxed* ordering provides no ordering guarantees. *Release-acquire* ordering establishes synchronization between a store and a load of the written value. *Sequentially consistent* ordering guarantees that the total order of accesses agrees with the program order in each thread. For a formal definition, see the C++ reference on ordering [8]. The stronger the guarantees are, the higher the performance cost.

3.5.1 Sequentially consistent ordering

First, we will look at how we can verify programs using atomics with sequentially consistent ordering. This provides very strong guarantees, and is thus easy to understand.

For verification, we will consider the atomic variables to be part of the environment. This may seem a bit strange at first, because they are internal to the program. But atomics allow communication between threads, which is not that much different from processes communicating over a network.

Sequential consistency means that there exists a total order of all accesses, such that each load reads the value of the latest preceding store to the same variable, and this order agrees with the program order. We can model this by storing the value of each atomic variable in the abstract state. Each access is a non-mover, which directly operates on the value in the abstract state. This means that the total order of atomic accesses also has to agree with the total order of ghost lock steps. We say that two orders agree, if their union is acyclic. There is nothing which prevents that, assuming that I/O operations are not reordered across atomic accesses.

We need a way to identify atomic variables, so we require the user to define an `AtomicId` enum. Then, we can represent atomics of type `AtomicU64` in the

```
trusted method SCAtomicU64::new(
    val: u64, id: AtomicId, gl: &mut GhostLock)
  requires ¬gl.did_act()
  requires ¬gl.state.atomic_allocated.contains(id)
  ensures result.id = id
  ensures gl.state = AbsState{
    atomic_allocated: old(gl.state).atomic_allocated.add(id),
    atomic_u64: old(gl.state).atomic_u64.set(id, val),
    ..old(gl.state)
  }

trusted method SCAtomicU64::load(&self, gl: &mut GhostLock): u64
  requires ¬gl.did_act()
  ensures result = gl.state.atomic_u64.get(self.id)
  ensures gl.state = old(gl.state)

trusted method SCAtomicU64::store(
    &self, val: u64, gl: &mut GhostLock)
  requires ¬gl.did_act()
  ensures gl.state = AbsState{
    atomic_u64: old(gl.state).atomic_u64.set(self.id, val),
    ..old(gl.state)
  }

trusted method SCAtomicU64::compare_exchange(&self,
    current: u64, new: u64, gl: &mut GhostLock): Result<u64, u64>
  requires ¬gl.did_act()
  ensures match result{
    Ok(v) ↦ v = current ∧ current = old(gl.state).atomic_u64.get(self.id) ∧
      gl.state = AbsState{
        atomic_u64: old(gl.state).atomic_u64.set(self.id, new),
        ..old(gl.state)
      }
    Err(v) ↦ v = gl.state.atomic_u64.get(self.id) ∧ gl.state = old(gl.state)
  }
```

Listing 3.10: Specifications for sequentially consistent atomics. Boilerplate pre- and postconditions for *locked*, *env_state* and *id* are omitted.

abstract state with a field *atomic_u*64: `AbsMap<AtomicId, u64>` (we need one such field for each of the atomic types). When we construct an `AtomicU64`, we provide the `AtomicId`, which is then stored in a ghost field, and used when performing operations. The *atomic_u*64 field is also part of the `EnvState`, so that it can only be modified by performing atomic operations.

However, it is unsound if multiple `AtomicU64` are constructed with the same `AtomicId`. This is because, if we store a value in one instance, and then load from another with the same id, the model tells us that we load the same value, but during execution the value is of course different. We could either just require the programmer to be careful and not do this. Or we can verify that it does not happen, by keeping track of which ids have been allocated in the abstract state, and only allowing instances to be created if the id has not been allocated. To do this, we can either add a new field *atomic_allocated*: `AbsSet<AtomicId>`, or change the type of the *atomic_u*64 field to `AbsMap<AtomicId, Option<u64>>`, where `None` indicates that the id is not allocated. In practice, the first option seems more convenient. Because we now need to prove that the id has not been allocated, the model action where the id is allocated needs to be protected by a guard.

Listing 3.10 shows the specifications for the atomic operations.

3.5.2 Release-acquire ordering

Sequential consistency is easy to reason about, but it has a performance cost. For performance-critical code, we therefore prefer to use release-acquire ordering, at the cost of more complex correctness arguments.

First, we look at how release-acquire ordering is formally defined [8]. *Happens-before* is a strict partial order of accesses. We have happens-before between subsequent accesses in the same thread (program order), and between a store and a load that reads from this store (synchronizes-with). There exists a *modification order* for each variable, which is a total order of stores to that variable. *Coherence* requirements guarantee that happens-before and modification order do not contradict:

1. Write-write coherence: If two stores to the same variable are ordered by happens-before, they are ordered the same way by modification order.
2. Read-read-coherence: If two loads from the same variable are ordered by happens-before, they either read from the same store, or the stores they read from are ordered the same way by modification order.
3. Read-write coherence: If a load happens-before a store to the same variable, then the load reads from an earlier store in modification order.
4. Write-read coherence: If a store happens-before a load from the same

variable, then the load reads from the same or a later store in modification order.

The release-acquire model is a declarative model, but for specifying the behavior of operations, we need an operational model. Lahav et al. [9] present such an operational model for a slightly strengthened version of release-acquire. In this model, there is a local memory and an outgoing message buffer for each processor. Processors read from and write to their local memory, and append writes, together with a global timestamp, to their message buffer. Non-deterministically, processors update their local memory from the message buffers of all other processors. However, this model does not seem practical for verifying programs.

Instead, we start with a very simple and weak model, which we will strengthen later: Each `AtomicU64` is represented by an `AbsSet<u64>` in the model. A store adds the value to the set, and a load returns any value in the set. This model is correct, because we know that for any load, the store that it reads from happens-before it. The happens-before order agrees with the program order and with the total order of sequentially consistent accesses, so nothing prevents it from also agreeing with the order of ghost lock steps. In this model, load is a right mover, because any loaded value can still be loaded later; and store a left mover, because any stored value could have been stored earlier without changing the values loaded by other threads. The model is already sufficient for some use cases, such as a flag that is only set once.

To strengthen the model, we will take advantage of the requirements on happens-before and modification order. But before we can even talk about these relations, we need a way to refer to the operations, which we do with `OpId`, a type alias for `Id`. This is just an opaque identifier; we used the same type for identifying ghost lock steps. Instead of directly keeping a set of values in the abstract state, we store an `AbsSet<OpId>`, containing the set of ids of all store operations which have been performed on the variable so far. To refer to the stored values, we introduce the abstract function *op_value*(*id*: `OpId`): `T`. The return value is generic to allow the function to be used for any of the primitive types. We also no longer need a separate field in the abstract state for each primitive type as we did for sequentially consistent atomics. The `store` method now returns the `OpId`, and `load` returns the `OpId` of the store that it reads from and its own `OpId`, in addition to the value.

We can now introduce an abstract function to represent the happens-before relation. For each load, the store that it reads from happens-before it; we can now state this as a postcondition on `load`. Happens-before also holds between subsequent operations in the same thread. To be able to express this, we introduce the `HappenedBeforeToken`, which is a struct containing an `OpId` in a private field. The meaning of this token is that the contained id happens-

before any operation performed in the thread where and when it exists. Each atomic operation returns a token containing its own `OpId`, and also takes a token in a parameter. We then establish that the id in the passed-in token happens-before the operation. This means that we can pass the token of an operation we performed earlier in the same thread to another operation, and we obtain happens-before between them. The `HappenedBeforeToken` can even be sent between threads, which is sound, because any way of passing values between threads must establish happens-before between the sending and receiving operation. Here we assume that the restriction on sending/receiving values describes in Section 3.4.6 have been implemented; otherwise we could obtain happens-before relations which contradict the ghost lock step order, which might be problematic. It must, however, not be possible to 'send' a token by putting it into the abstract state, which we prevent by not deriving the `Copy` trait for `HappenedBeforeToken`. The `AbsState` must be `Copy` and hence cannot contain a token.

Essentially, we are building a relatively simple operational model which says that each load reads from a store that has happened earlier in ghost lock step order, and then putting the declarative model that defines release-acquire on top of this. We could then use the coherence requirements to further restrict the set of stores that each load can read from. However, we have not implemented this.

For a successful compare-and-exchange operation, we know that it reads from the immediately preceding operation in modification order. We represent this with the abstract function $mo\_next(id\colon \texttt{OpId})\colon \texttt{OpId}$, which assigns to an operation the immediate successor in modification order. After a successful compare and exchange, we then learn that mo_next of the operation that it reads from is the compare-and-exchange operation itself. This is already sufficient for verifying our case study which uses atomics. In fact, we did not even need the happens-before relation for this.

3.6 Modularity

We would like to create reusable modules which can use the ghost lock. For example, we would like to create an atomics module, which wraps the atomics of the Rust standard library, and provides trusted specifications of how actions on the atomics interact with the abstract state in the ghost lock. Each program could then use the subset of modules that it needs in its model.

Moreover, we would like to create modules which are verified instead of trusted. For example, our hash set case study (Section 4.2) is a data structure which is verified against a model, but is not useful on its own, as it needs to be integrated into a program which uses the hash set. We would like to use

the hash set as a module in a program with its own model, and somehow link the program's model with the hash set's model.

The hash set is a verified module, which in turn uses the atomics module. Larger applications could be composed of a hierarchy of independently verified modules.

3.6.1 Trusted modules

To make the atomics module reusable, it cannot have access to the concrete types of a specific model, such as `AbsState`. But if the module does not know the concrete type of `AbsState`, it also cannot access the *atomics* field inside that struct. Therefore, we *project* the `GhostLock` to a `ProjectedGhostLock`, which only contains the *atomics* part of the `AbsState`.

This is done by the `project` method, which takes a **&mut** `GhostLock` and returns **&mut** `ProjectedGhostLock`. For the specification of this method, the user has to provide two functions:

function *get_module*(*s*: `AbsState`): `AtomicsState`
  s.atomics

function *put_module*(*s*: `AbsState`, *atomics*: `AtomicsState`): `AbsState`
  `AbsState`{*atomics*: *atomics*, ..*s*}

With this, we can specify that the *state* field of the projected ghost lock is initially extracted from the `AbsState` with *get_module*, and once the **&mut** `ProjectedGhostLock` expires, we put the new value back into the `AbsState` with *put_module*. For providing specifications that apply after expiry, we use the pledge feature of Prusti.

In the Rust implementation, both the regular and projected ghost lock are generic. The regular ghost lock is generic over a type which must implement the `Model` trait. In this trait, the `AbsState`, `Action`, `GuardKind` and `EnvState` must be defined as associated types, and *init*, *next*, *action_requires_guard* and *get_env_state* as associated functions. The type which implements this trait can be an empty struct. The projected ghost lock is generic over a type which must implement the `Module` trait. This trait only requires the `ModuleState` associated type. If we want to use a module `X` in a model, we then have to implement the trait `Project<X>` for the model type. This trait requires the two functions *get_module* and *put_module*.

The `ProjectedGhostLock` is reduced compared to the `GhostLock`; it only has the *state* and *did_act* field. It is always implicitly locked, and its *state* field is private since it is implicitly part of the environment state. The module can now provide trusted specification for how its methods interact with the state of the module, and whether these methods are left movers, right movers or

non-movers. The program passes `project`(**&mut** gl) to the methods of the module.

3.6.2 Modules with inner model

The hash set model has various fields to represent internal state, but at the highest level we have the *set*: `AbsSet<u64>` field, which represents the hash set as an abstract set. A program which uses the hash set is not interested in the internal state, but only in how operations interact with this abstract set. We would like to *link* the *set* field of the hash set's model to a *set* field of the same type in the program's model. The program could then use the hash set module much like the atomics module, with operations having specifications of how they interact with the abstract set. But unlike the trusted specifications of the atomics module, these specifications are verified. They form a contract, which is fulfilled by the module and relied on by the program.

The idea is that the models of the program and of the hash set are mostly independent, except that the *set* fields of both are linked; they always have the same value and are updated together. We can represent this as a single model which is a composition of the two models. We have model a, with `AbsStateA`, `ActionA`, $init_a$, $next_a$, and analogously for model b. The composed model then looks as follows:

struct `AbsState` = $\{a$: `AbsStateA`, b: `AbsStateB`$\}$

enum `Action` = `A`(`ActionA`) | `B`(`ActionB`) | `Both`(`ActionA`, `ActionB`)

ghost function $init$(s: `AbsState`): `bool`
  $init_a(s.a) \land init_b(s.b) \land s.a.set = s.b.set$

ghost function $next$(s: `AbsState`, n: `AbsState`, act: `Action`): `bool`
  match act {
    `Action::A`(a) $\mapsto$ $next_a(s.a, n.a, a) \land n.a.set = s.a.set \land n.b = s.b$
    `Action::B`(b) $\mapsto$ $next_b(s.b, n.b, b) \land n.b.set = s.b.set \land n.a = s.a$
    `Action::Both`(a, b) $\mapsto$
      $next_a(s.a, n.a, a) \land next_b(s.b, n.b, b) \land n.a.set = n.b.set$
  }

Both *set* fields must be initialized with the same value. A step can be in just one of the two model, but then the *set* field cannot change. It can only change when performing a step in both models simultaneously, and in that case both fields must be changed to the same new value. Note that this combined model is only for illustration, it does not exist in the implementation.

To implement this idea, we extend the `ProjectedGhostLock` with a *ig* field, short for 'inner ghost lock', which contains the `GhostLock` for the inner model. The `Module` trait gets a new associated type `InternalModel`, which

is the model for this inner ghost lock. `InternalModel` must implement `Project<Self>`, such that we can project it to its own `ModuleState`.

The inner ghost lock is a full ghost lock, and is initially not locked. The module can use this like a regular ghost lock to make internal steps which are not linked to the outer ghost lock. The outer ghost lock remains locked during the entire time that the projected ghost lock exists. If the module wants to link an internal ghost lock step to the ongoing outer ghost lock step, it calls the `link` method on the `ProjectedGhostLock`. This method establishes that the value of the linked fields is equal between the inner and outer ghost lock. After calling `link`, the module can then call `linked_set` to simultaneously update the linked field in the inner and outer ghost lock.

Several things need to be considered to ensure that this is sound. The linearization point of the inner and outer step should coincide, since in the combined model, it really is a single step. We do this by making `link` a non-mover in the outer model, while it is a both-mover in the inner model. The linearization point of the outer model is not the `link` call itself, but equal to the linearization point of the inner model. The fact that `link` is a non-mover also ensures that the outer step can only be linked to a single inner step. It must only be possible to call `linked_set` if `link` has been called before in the same inner step, since the linked value can only be updated in a combined step. To ensure this, we use the abstract function $\mathit{linked}(\mathit{id}\colon \texttt{Id})\colon \texttt{bool}$, which is true if `link` has been called while the inner ghost lock had this id. The linked field should be part of the `EnvState` in both the inner and outer model, such that `linked_set` is the only way to update it.

There are still some unresolved soundness issues. We need to ensure that the inner ghost lock is initialized, and that the initial value of the linked field is the same in inner and outer model. For now, we do this ad-hoc by creating a `GhostLock` instance for the inner model, which is thrown away (since we obtain it later with `project`), and asserting that the linked field has the same value. Because we allow steps in the inner ghost lock which are not linked to the outer step, we can have two independent, concurrent ghost lock steps in the same thread, so considerations similar to allowing reentrancy apply, which we discussed in Section 3.3.6. There is also the issue that there could be multiple independent instances of the same model in a bigger application composed of multiple modules, which themselves rely on other modules. In this case, we have to ensure that these instances are not mixed. For example, guards or *old_state* facts from one instance should not be used on a different instance, as this would be unsound. We could address this by introducing an instance id, which we would attach to ghost locks, guards and facts, such that they cannot be mixed between instances. Alternatively, we might be able to solve this using the type system, by somehow making sure that the different instances have incompatible types.

We would like to use the same `ProjectedGhostLock` also for trusted modules which do not have an internal model. We can do so by using a dummy model, and adding trusted specifications to methods as before.

3.7 Assumptions

There are several assumptions that we make, which are needed for soundness, but not verified automatically. This means that the user of the approach has to check these assumptions manually or justify them with an external proof. Here we discuss these assumptions and how they could be lifted.

3.7.1 Initialization

The model is initialized by creating ghost lock and guard instances using the `GhostLock::new`(*init_state*) and `Guard::new`(*kind*) methods. If the model has constants, we also need to define their values. We make several assumptions about how these methods are used:

1. In each thread, at most one ghost lock instance is created.
2. Each ghost lock instance is created with the same initial state.
3. At most one guard is created of any given kind.
4. Model constants are set to the same value in each process.

The first assumption is needed because we disallow reentrancy, as discussed in Section 3.3.6. The next two assumptions are because the model can only have one initial state, and we assume that each guard is unique. Constants were introduced in Section 3.2.4; it would be a contradiction if a constant were set to two different values.

Lifting these assumptions is challenging, because the system in which the model is used can for example consist of distributed processes communicating over a network. We need to trust the user that all these processes are started with the correct configuration; for example, that each process has a unique identifier, such that guards created in each process with this identifier do not overlap.

What we can do is to create an *initialization model*, which is a verification-only method which simulates how the system is configured and initialized. The creation of processes can be simulated by spawning threads. We can make a wrapper for the thread spawn method which passes a ghost lock instance to the thread closure; this way we do not need to call the ghost lock constructor on each thread.

To verify that guards are unique, we introduce the *guard dispenser*. This is a struct which is used to create guard instances. It remembers which guards have been created, and only allows each kind to be created once.

With this initialization model, we can now lift the above assumptions, and we only have to check manually that the ghost lock and guard dispenser are only created once, and each constant is defined once. Of course, we have the new assumption that the initialization model is an accurate representation of the real initialization process, but at least it should be easier to compare the two.

3.7.2 Termination checking

After acquiring the ghost lock, the program can perform arbitrary actions, only restricted by the mover requirements. Only when the ghost lock is released do we check whether this sequence of actions is allowed by the model, and if not, verification will fail at that point.

This assumes that, whenever the ghost lock is acquired, it is eventually released. There are two ways to violate this assumption: Either by blocking or looping infinitely, or by exiting the thread or process without releasing the ghost lock. This is also related to the second condition of the D-reduction (Section 2.2), which we have ignored so far. It requires all except the first action of a ghost lock step to always be able to execute, i.e., they cannot block.

Maier [10] describes how to implement termination checking in Prusti, which is exactly what we need here, but this is not yet available in the version of Prusti used for the evaluation of this work. The feature allows us to attach a `terminates` attribute to a method, which then requires that method to terminate. The question is now, which method do we attach the attribute to? We cannot just attach it to the entry method of the program, since the program may not terminate. Instead, we could imagine changing the interface for making ghost lock steps, such that we call some method with a closure, which performs the step and returns the performed `Action`. We would then require this closure to terminate.

To prevent exiting a thread without releasing the ghost lock, our wrapper method of spawn can require the ghost lock to be in unlocked state when the thread closure returns. By not adding the `terminates` attribute to the `process::exit` method, we can also block this way of avoiding to release the ghost lock.

Chapter 4

Evaluation

4.1 Case study: Paxos

Our first case study is a verified implementation of the Paxos consensus algorithm [11]. It is based on the TLA$^+$ model of Paxos available at [12] in the accompanying material. Table 4.1 shows some statistics for this case study.

The Paxos consensus algorithm lets a set of processes choose a value by communicating with each other over a network. The algorithm guarantees that, once a value is chosen, no other value can be chosen. There are three roles: *proposers*, *acceptors* and *learners*. At any point, a proposer can start a *ballot* by sending a 1A message to all acceptors, containing a ballot number. When receiving this message, each acceptor checks whether it has seen a bigger ballot number before, and if not, it replies with a 1B message, containing its identity, the received ballot number, and the value and ballot number of the latest vote it has made so far, if any. The 1B message represents a promise not to vote in a ballot with smaller than the received number. The proposer then collects 1B messages, until it has received one from a quorum

	Model	Impl	Proof	Verification time
Complete	112	132	152	3.5m
Phase 1A	4	14	24	30s
Phase 2A	21	50	61	2.5m
Phase 1B+2B	27	33	40	1m
Shared part	60	35	27	30s

Table 4.1: Statistics for the Paxos case study. We show statistics for the complete code, and also for individual phases of the algorithm, and the part shared between phases. For each of these, we show lines of code for model, implementation and proof, and verification time. Implementation includes only the lines that are necessary to execute the program. Proof includes pre- and postconditions, loop invariants, assertions, and ghost code.

$$
\begin{aligned}
\mathit{TypeOK} \triangleq\ & \land \mathit{maxBal} \in [\mathit{Acceptor} \to \mathit{Ballot} \cup \{-1\}] \\
& \land \mathit{maxVBal} \in [\mathit{Acceptor} \to \mathit{Ballot} \cup \{-1\}] \\
& \land \mathit{maxVal} \in [\mathit{Acceptor} \to \mathit{Value} \cup \{\mathit{None}\}] \\
& \land \mathit{msgs} \subseteq \mathit{Message}
\end{aligned}
$$

$$
\mathit{Send}(m) \triangleq \mathit{msgs}' = \mathit{msgs} \cup \{m\}
$$

$$
\begin{aligned}
&\mathit{Phase1b}(a) \triangleq \\
&\quad \land \exists\, m \in \mathit{msgs} : \\
&\qquad \land m.\mathit{type} = \text{“1a”} \\
&\qquad \land m.\mathit{bal} > \mathit{maxBal}[a] \\
&\qquad \land \mathit{maxBal}' = [\mathit{maxBal} \text{ EXCEPT } ![a] = m.\mathit{bal}] \\
&\qquad \land \mathit{Send}([\mathit{type} \mapsto \text{“1b”}, \mathit{acc} \mapsto a, \mathit{bal} \mapsto m.\mathit{bal}, \\
&\qquad\qquad\qquad \mathit{mbal} \mapsto \mathit{maxVBal}[a], \mathit{mval} \mapsto \mathit{maxVal}[a]]) \\
&\quad \land \text{UNCHANGED } \langle \mathit{maxVBal}, \mathit{maxVal} \rangle
\end{aligned}
$$

Listing 4.1: The phase 1b step of the TLA$^+$ model of Paxos [12].

of acceptors. If none of these acceptors had voted before, the proposer is free to choose a value, otherwise it must take the value from the vote with largest ballot number in the received messages. It then sends a 2A message to all acceptors, containing the ballot number and value. Upon receiving this message, each acceptor again checks whether it has seen a bigger ballot number, and if not, it votes for this value, by sending a 2B message to all learners, containing its identity, the ballot number and value. When a learner has received the same vote from a quorum of acceptors, it learns that this value has been chosen.

We will now look at phases 1B and 2A of the algorithm in more detail. We discuss how we translate these parts of the original TLA$^+$ model to Rust syntax, and how we implement the phases. The other two phases are not discussed, as there is nothing new to see there.

4.1.1 Phase 1b

Phase 1B is the step where an acceptor sends a 1B message.

Listing 4.1 shows this step in the the original TLA$^+$ specification. The model has four variables; *TypeOK* shows their types. $\mathit{maxBal}[a]$ is the largest ballot number that acceptor a has seen. $\mathit{maxVal}[a]$ and $\mathit{maxVBal}[a]$ are the value and ballot number of the latest vote cast by a. Finally, *msgs* is the set of messages that have been sent. In the *Phase1b* step, we check that a message of type 1A has been sent, and that the ballot number is the biggest we have seen so far. We then update $\mathit{maxBal}[a]$ and send the 1B message.

Listing 4.2 shows the same step after translating it to Rust as described in Section 3.2. For the state, we use `Option` instead of sentinel values to denote

```
struct AbsState {
  msgs: AbsSet<Message>
  max_bal: AbsMap<Acceptor, Option<Ballot>>
  max_vote: AbsMap<Acceptor, Option<(Ballot, Value)>>
}

ghost function next(s: AbsState, n: AbsState, act: Action): bool
  match act {
    Action::Phase1b(a, b) ↦
      s.msgs.contains(Message::Msg1a(b)) ∧
      match s.max_bal.get(a) {
        Some(mb) ↦ ballot_greater(b, mb)
        None ↦ true
      } ∧
      n = AbsState{
        msgs: s.msgs.add(Message::Msg1b(a, b, s.max_vote.get(a))),
        max_bal: s.max_bal.set(a, Some(b)),
        ..s
      }
    ...
  }
```

Listing 4.2: The phase 1B step of the model translated to our syntax.

that no ballot has been seen or no vote cast yet. We also merge *maxVBal* and *maxVal* into the single *max_vote* field. In the `Phase1b` step, we avoid the existential quantifier for the message by adding the ballot number to the action parameters. Ballot numbers are compared using the *ballot_greater* function; this is because our ballot numbers are actually a pair of a number and the proposer identifier. We do this because each proposer must have its own space of ballot numbers.

The implementation of phase 1B is shown in Listing 4.3. `run_acceptor` is the method which runs in each acceptor process. In a loop, it receives messages and processes them.

The *a* parameter is this acceptor's identity. Each acceptor has a guard, which protects the `Phase1b` and `Phase2b` actions for this acceptor. This means that the entries of the *max_bal* and *max_vote* fields that belong to this acceptor can only be changed with this guard. We keep copies of these values in the local variables of the same name. After opening the guard, we can then learn that the local variables match the values in the abstract state. We always update both at the same time, such that they stay in sync; in phase 1B we see this in the *max_bal* update.

Message sending and receiving works the same as with the Pong server in

```
method run_acceptor(a: Acceptor, gl: &mut GhostLock,
    guard: &mut Guard, init_state: AbsState){
  let mut max_bal: Option<Ballot> := None;
  let mut max_vote: Option<(Ballot, Value)> := None;
  let mut guard_state := init_state;
  loop{
    invariant ¬gl.locked();
    invariant guard.kind() = GuardKind::Acceptor(a);
    invariant released(guard.last_id());
    invariant final_state(guard.last_id()) = guard_state;
    invariant guard_state.max_bal.get(a) = max_bal;
    invariant guard_state.max_vote.get(a) = max_vote;
    gl.acquire();
    let msg := receive(gl);
    let receive_state := gl.state;
    gl.release_stutter();
    match msg {
      Message::Msg1a(b) ↦ {
        if match max_bal { Some(mb) ↦ ballot_greater(b, mb)
                           None ↦ true }{
          gl.acquire();
          gl.preserved_predicate(receive_state,
            λs. receive_state.msgs.is_subset(s.msgs));
          guard.open(gl);
          guarded_preserved_predicate(
            guard_state, gl.state, guard.kind(),
            λs. s.max_bal.get(a) = max_bal ∧
                s.max_vote.get(a) = max_vote);
          max_bal := Some(b);
          gl.state.max_bal := gl.state.max_bal.set(a, max_bal);
          send(Message::Msg1b(a, b, max_vote), gl);
          guard_state := gl.state;
          gl.release(Action::Phase1b(a,b));
        }
      }
      ...
    }
  }
}
```

Listing 4.3: The implementation of phase 1B. Preconditions for `run_acceptor` are omitted, they are the same as the loop invariants except with *init_state* instead of *guard_state*.

$$
\begin{array}{l}
Phase2a(b, v) \triangleq \\
\quad \land \neg \exists\, m \in msgs \quad : m.type = \text{“2a”} \land m.bal = b \\
\quad \land \exists\, Q \in Quorum : \\
\qquad \textsc{let}\ Q1b \triangleq \{m \in msgs \ : \land m.type = \text{“1b”} \\
\qquad\qquad\qquad\qquad\qquad\quad \land m.acc \in Q \\
\qquad\qquad\qquad\qquad\qquad\quad \land m.bal = b\} \\
\qquad\qquad Q1bv \triangleq \{m \in Q1b : m.mbal \geq 0\} \\
\qquad \textsc{in} \quad \land \forall a \in Q : \exists\, m \in Q1b : m.acc = a \\
\qquad\qquad \land \lor Q1bv = \{\} \\
\qquad\qquad\quad \lor \exists\, m \quad \in Q1bv : \\
\qquad\qquad\qquad\quad \land m.mval = v \\
\qquad\qquad\qquad\quad \land \forall\, mm \in Q1bv : m.mbal \geq mm.mbal \\
\quad \land Send([type \mapsto \text{“2a”}, bal \mapsto b, val \mapsto v]) \\
\quad \land \textsc{unchanged}\ \langle maxBal, maxVBal, maxVal \rangle
\end{array}
$$

Listing 4.4: The phase 2A step of the TLA$^+$ model of Paxos [12].

Listing 3.6. We use `preserved_predicate` to establish that the *msgs* field of the state at the time we received the message is a subset of $gl.state.msgs$. In particular, this means that $gl.state.msgs.contains(msg)$.

4.1.2 Phase 2a

Phase 2A is the most complex of the four phases of Paxos. As in the previous section, the original TLA$^+$ model is shown in Listing 4.4, and our translated version in Listing 4.5.

In this phase, we have to prove that we received a 1B message for the ballot from each of a quorum of acceptors. In the original TLA$^+$ model, the relevant messages are first filtered from *msgs* using a set comprehension and bound to $Q1b$. It requires there to be a message from each acceptor of the quorum in $Q1b$. The subset of messages is then further filtered to $Q1bv$, containing only messages with votes. This is then used to specify the requirement that if there are votes, the picked values must be from the vote with the largest ballot number.

In our translation of the step, we avoid set comprehensions. Instead of the disjunction and existential quantifier for the maximum vote, we use the action parameter max_b, which contains the acceptor and ballot number of the maximum vote, if any.

One particular difference between the two versions is that in the original step, we have to prove that all 1B messages in *msgs* of ballot b have votes with number less or equal to the claimed maximum vote. In other words, we have to prove that there do *not* exist messages in *msgs* where the number is bigger. In our version on the other hand, only the existence of messages

```
Action::Phase2a(b, v, q, max_b) ↦
  ¬∃v2. s.msgs.contains(Message::Msg2a(b, v2)) ∧
  quorum().contains(q) ∧
  match max_b {
    None ↦ ∀a. q.contains(a) →
      s.msgs.contains(Message::Msg1b(a, b, None))
    Some((a, mb)) ↦ q.contains(a) ∧
      s.msgs.contains(Message::Msg1b(a, b, Some((mb, v)))) ∧
      ∀a2. q.contains(a2) → ∃vote.
        s.msgs.contains(Message::Msg1b(a2, b, vote)) ∧
        match vote {
          Some((mb2, _)) ↦ ballot_greater_equal(mb, mb2)
          None ↦ true
        }
  } ∧
  n = AbsState{ msgs: s.msgs.add(Message::Msg2a(b, v)), ..s }
```

Listing 4.5: The phase 2A step of the model translated to our syntax. We omit the surrounding *next* function and match expression.

has to be proven. This is relevant, because in the implementation, we cannot directly prove the absence of messages. All we know about *msgs* is that it contains the messages that we received. It would still be possible to implement phase 2A with the original version of the model though. We know that each acceptor sends at most one 1B message for any ballot number, and could take advantage of this using `preserved_predicate` to prove that there cannot exist other messages from the same acceptors with the same ballot number. By writing the model in such a way that only presence of messages that were received has to be proven, we can avoid this reasoning step in the implementation. This change does not affect the possible behaviors of the model, though. As we saw before, the additional messages whose absence we do not have to prove in the new version could not exist anyway.

In our version of the model, we have replaced most existential quantifiers with parameters on the action, but not all of them. We experimented with replacing the $\exists vote$ quantifier in our version with a parameter of type `AbsMap<Acceptor, Option<(Ballot, Value)>>`, but this did not improve verification time. Usually though, replacing quantifiers with parameters does help verification to succeed.

The implementation of phase 2A receives messages in a loop, until a 1B message is received from a majority of acceptors. We need to remember from which acceptors we already received a message to avoid duplicates, so we store this in a hash set. We also keep an `AbsSet` version of this set, which is later used as the q parameter of the `Phase1a` action. The maximum

Model	Impl	Proof	Verification time
123	57	376	3m

Table 4.2: Statistics for the hash set case study. The unverified implementation has 43 lines of code. After verification, this has slightly expanded as some parts had to be split into multiple lines.

vote we have seen so far is kept in a variable, and we use a loop invariant to remember that it is the maximum vote. The proof of this loop invariant is non-trivial, since it requires transitivity of *ballot_greater_equal* when we find a vote with bigger number. Despite that, the verifier is able to prove it without additional help.

4.2 Case study: Lock-free hash set

Our second case study is a lock-free hash set, which allows multiple threads to concurrently insert values, and uses open addressing for collision resolution. It is based on the TLA$^+$ model and corresponding Java implementation available at [13]. Almost exactly the same hash set was part of the VerifyThis competition 2022 [14]. Table 4.2 shows statistics for this case study.

In addition to verifying that our implementation correctly implements our model, we also verify that the operations on the set behave like the corresponding operations on an abstract set. In Section 3.6.2, we discussed how we expose this to the user of the hash set. This means that we do not have to trust that the model is correct. We also implemented unit tests for the hash set, to check that it is executable and works as expected.

To give some context, this is a special-purpose data structure used in the TLC model checker [15]. The model checker walks the graph of reachable states, starting from the initial state, and for that, it needs to know which states it has already visited. To do that, it computes a 64-bit hash of each found state, which is called a *fingerprint*, and then tries to insert it into the hash set. If the fingerprint was already in the set, this state has already been visited (or a different state with the same hash, i.e., a collision). Otherwise, the state is inserted into a queue for later processing. This walking of the state graph is done on multiple threads in parallel, so the hash set needs to support concurrent accesses.

The hash set consists of an array of 64-bit integers, which are accessed with atomic operations. Initially, each array entry is zero, indicating that it is empty. A non-zero array entry means that the set contains this value. The set cannot contain the value 0. Values cannot just be inserted at an arbitrary position in the array; instead, the 64-bit value is scaled linearly to the length of the array to obtain the primary position. When inserting, we first check

```
enum AtomicId = ArrayEntry(usize)

struct AbsState {
  set: AbsSet<u64>
  array: AbsMap<usize, u64>
  array_len: usize
  val_to_probe: AbsMap<u64, usize>
  idx_to_init_op: AbsMap<usize, OpId>
  idx_to_last_op: AbsMap<usize, OpId>
  atomics: AbsMap<AtomicId, AbsSet<OpId>>
}

enum Action = Init(usize, OpId) | Add(u64, usize, OpId, OpId)
```

Listing 4.6: The state and action types of our hash set model.

the primary position. If it already contains a different value, we check the primary position + 1, wrapping around at the end of the array. We continue looking for an empty position, up to the *probe limit*, which is the maximum number of positions that we check before giving up; by default this is 1024. The value must be inserted in the first empty position seen while probing. This means that, when we find an empty position and have not seen the value yet, we can be sure that the set does not already contain the value. The hash set only supports a single operation, which is to insert values; it is not possible to remove values.

In the full implementation, we do not just give up when we reach the probe limit. Instead, we lock the set and wait until all other threads finish what they were doing. Then, all values from the array are flushed into a file, containing sorted fingerprints, and work can continue. For each insert operation, we now also need to check whether the fingerprint is contained in the file. We omit this part in our verified implementation.

4.2.1 Model

Our model is very different from the original TLA$^+$ model [13] of the hash set. The original model describes flushing to disk, which we omit. It is written in the PlusCal language, which allows to write TLA$^+$ models in an imperative style. It is focused more on *how* the algorithm works and has many steps, while our model only describes *what* it does, and leaves the how to the implementation.

The state of our model is shown in Listing 4.6. At the highest level of abstraction, we have the *set* field, which contains the set of values in the hash set. At a lower level, we have the *array* field, which contains the current value of each array entry. *array_len* is the length of the array, this stays constant.

For each value in the set, we store the probe offset where it is stored in the array in *val_to_probe*. At the lowest level of abstraction, we have the *atomics* field. We use our release-acquire model for the atomics as described in Section 3.5.2, where the atomic operations are specified in terms of this field. There is an atomic variable for each array entry. The *atomics* field contains the set of store operations for each atomic variable, identified by the `OpId`. We also separately store the individual operations that we perform in the *idx_to_init_op* and *idx_to_last_op* fields.

In Listing 4.7, we show initialization and transition relations of our model. We only care about the initial values for *set*, *array* and *atomics*. The value of $val\_to\_probe.get(val)$ for a given *val* only matters if *val* is contained in *set*, so the initial value of *val_to_probe* is irrelevant; and something similar applies to *idx_to_init_op* and *idx_to_last_op*.

There are only two actions in our model: `Init` initializes an atomic variable, and `Add` inserts a value into the set. The `Init` action is protected by a guard, because we can only initialize a variable if we know that it is not already initialized. Other than initializing the atomic variable with the value *EMPTY*, this action only stores the initialization operation in *idx_to_init_op*, so that we can later refer to it.

The `Add` action is the most interesting, and there is a lot going on here. *op* is the atomic operation which writes the value into the array. We first check that the value stored by this operation is *val*, and that this value is non-zero. The array length must be non-zero, since this is a precondition of *get_idx*, which computes the position in the array for a given value and probe offset. *probe* contains this offset, and we check that every array entry before this is already filled. The entry itself must be empty. *op* is actually a compare-and-exchange operation, which atomically replaces the previous empty value with *val*. To check this, we require the operation immediately following the initialization operation in modification order to be *op*. Finally, we also check that the atomic variable has been initialized. If all these conditions are satisfied, we then update the state accordingly.

4.2.2 Implementation

The constructor creates the array and initializes the atomic variables in a loop, performing an `Init` action for each. The `insert` operation has an outer loop which searches the first empty probe offset, remembering that $entry\_filled$ holds for every previous position. In an inner loop, we read the value from the array, and if empty, try to compare and exchange it with the value to be inserted. The compare-and-exchange operation may fail in case of concurrent operations by other threads, in which case we have to try again. We may also find that the value is already present in the set; in which case we indicate this fact to the caller.

```
const EMPTY: u64 := 0

ghost function init(s: AbsState): bool
  s.set = AbsSet::empty() ∧
  s.array = AbsMap::new(EMTPY) ∧
  s.atomics = AbsMap::new(AbsSet::empty())

ghost function entry_filled(s: AbsState, val: u64, probe: usize): bool
  s.array.get(get_idx(s.array_len, val, probe)) ≠ val ∧
  s.array.get(get_idx(s.array_len, val, probe)) ≠ EMPTY

ghost function next(s: AbsState, n: AbsState, a: Action): bool
  match a {
    Action::Init(idx, op) ↦
      let id := AtomicId::ArrayEntry(idx) in
      op_value(op) = EMPTY ∧
      s.atomics.get(id) = AbsSet::empty() ∧
      n = AbsState{
        idx_to_init_op: s.idx_to_init_op.set(idx, op),
        atomics: s.atomics.set(id, AbsSet::singleton(op)),
        ..s
      }
    Action::Add(val, probe, init_op, op) ↦
      op_value(op) = val ∧ val > 0 ∧ s.array_len > 0 ∧
      ∀p. p < probe → entry_filled(s, val, p) ∧
      let idx := get_idx(s.array_len, val, probe) in
      let id := AtomicId::ArrayEntry(idx) in
      s.array.get(idx) = EMPTY ∧
      s.idx_to_init_op.get(idx) = init_op ∧
      mo_next(init_op) = op ∧
      s.atomics.get(id) ≠ AbsSet::empty() ∧
      n = AbsState{
        set: s.set.add(val),
        array: s.array.set(idx, val),
        val_to_probe: s.val_to_probe.set(val, probe),
        idx_to_last_op: s.idx_to_last_op.set(idx, op),
        atomics: s.atomics.set(id, s.atomics.get(id).add(op)),
        ..s
      }
  }
```

Listing 4.7: The *init* and *next* relations of our hash set model.

```
method insert(&self, val: u64, gl: &mut ProjectedGhostLock):
    Result<bool, ()>
  requires val ≠ EMPTY
  requires self.invariant()
  requires ¬gl.ig.locked()
  ensures ¬gl.ig.locked()
  requires ¬gl.did_act()
  ensures match result {
    Ok(p) ↦ old(gl.state()).contains(val) = p ∧
      gl.state() = old(gl.state()).add(val)
    Err(()) ↦ ¬gl.did_act() ∧ gl.state() = old(gl.state())
  }
```

Listing 4.8: The specification of the `insert` method.

In Listing 4.8, we show the specification of the `insert` method. It returns a Boolean to indicate if the value was already in the set. Insertion may fail if the probe limit is reached, in which case an error is returned. The high-level operation is specified using the `ProjectedGhostLock`.

In the implementation, we make heavy use of `preserved_predicate` to prove all kinds of facts about the abstract state. Many of these calls are in separate lemma methods which are called from the `insert` method. Splitting it up this way makes it easier to read, but also improves verification time.

As a simple example; if we load an array entry and find that the value is equal to the value we are trying to insert, we want to conclude that the value is already in the set. When we perform the load operation, we obtain the operation *prev_op* which has stored the value we loaded, and we learn that $gl.state.atomics.get(id).contains(prev\_op)$ and $val = op\_value(prev\_op)$. We can use `preserved_predicate` with the initial state and the predicate $\lambda s.\ s.atomics.get(id).contains(prev\_op) \rightarrow s.set.contains(val)$ to obtain that $gl.state.set.contains(val)$. This works because the step which added *prev_op* to *atomics* must also have added *val* to the set. Notably, we do not have to explicitly state all the preconditions that are required for the predicate to be preserved. The verification tool can use facts such as $val = op\_value(prev\_op)$ from the context for the proof.

For some more complex reasoning steps, we call `preserved_predicate` with a predicate of the form $\lambda s.\ \forall x.\ p(s, x)$, where p is substituted with some formula. To verify this, the verification tool will need to prove

$$\forall s, n, a.\ (\forall x.\ p(s, x)) \wedge next(s, n, a) \rightarrow (\forall x.\ p(n, x)),$$

which it can have trouble with. We found that in some cases, we could make it work by asserting $\forall s, n, a, x.\ p(s, x) \wedge next(s, n, a) \rightarrow p(n, x)$ before the call

to `preserved_predicate`. For the cases where p is more complex, we created a lemma method which takes a predicate p and has the formula from the assertion mentioned before as a precondition, but now without p substituted for a different formula. We can directly use `preserved_predicate` with $\lambda s.\ \forall x.\ p(s, x)$ to prove this lemma method.

4.3 Comparison with other refinement approaches

4.3.1 Flexible Refinement Proofs in Separation Logic

Our approach is based on the refinement approach described by Bílý et al. [1], and the accompanying examples written in the Viper language. The high-level ideas are similar to our approach, but the encoding used in the Viper examples is quite different, especially for guards.

Guards are represented as Viper permissions. To define which actions require which guards, they require the guard permission to be present in the preconditions of the action itself. Guard-preserved predicates are not provided inline in the implementation as in our approach, but must be stated as separate Boolean functions. For each such function, there is a method in which they prove that its value is preserved by the *next* relation if the permission for this guard is absent.

In our approach, on the other hand, we separate the identification of guards (`GuardKind`) from the guard resource itself (`Guard`). To define which actions require which guards, we have the separate *action_requires_guard* function. We rely heavily on abstract functions to represent the state of ghost lock steps, such as whether a guard has been opened. As we saw in Section 4.2.2, the fact that we state (guard-)preserved predicates inline in the implementation instead of separately means that we can rely on facts from the context without having to explicitly state them.

4.3.2 IronFleet

IronFleet [16] is another approach which can prove refinement between a TLA$^+$-style model and an implementation. It is implemented in the Dafny language, which has similar features as Prusti. This approach is specifically for the verification of distributed systems, consisting of a set of hosts communicating over a network.

As in our approach, they define a global state machine in terms of an *init* and *next* predicate. Unlike our approach, this is then first refined to a distributed state machine, with state consisting of the environment state and the state of each host. A step in this state machine is a step of one of the hosts.

The implementation is provided in the form of two methods `HostInitImpl` and `HostNextImpl`. `HostNextImpl` takes a `HostEnvironment` and `HostState`, and must return the new `HostState`. This method then runs in a loop on each host. In our approach, this could be represented as a loop in which we acquire the ghost lock, call `HostNextImpl` and release the ghost lock. The `HostEnvironment` records a history of environment interactions, and serves the same role as the *env_state* field of our ghost lock. Rather than having ghost state that the implementation can modify (the *state* field in our approach), we have an abstraction function which takes the `HostState` and returns the corresponding state of the host model. There are no guards in this approach; each host can only modify its host state and the environment state.

There is nothing like the `preserved_predicate` method of our approach. The host *next* relation corresponds closely to the implementation and has no preconditions that would require this. Instead, they use invariants of the model for the refinement proof between the global and distributed models.

As in our approach, they use a reduction argument to allow a single step to perform a sequence of receives followed by a sequence of sends. But they check this by stating it as a requirement on the history of environment interactions, rather than with a Boolean flag as in our approach.

4.3.3 Verus Transition Systems

Verus Transition Systems [17] is an approach intended mainly for the verification of multi-threaded code. They can define a state machine and then verify that an implementation refines it. It is part of Verus [18], a verification tool for Rust.

Rather than defining *init* and *next* relations as predicates, they use a custom language, where actions are written in an imperative style. In the implementation, they annotate atomic operations with the corresponding action of the state machine, using the `atomic_with_ghost` macro. This corresponds to our ghost lock step. They do not use a reduction argument, and instead only perform one atomic operation per state machine step.

They use *tokens*, which are linked to a specific field of the state, rather than guards which protect certain actions. There are different *tokenization strategies*. For example, in the *variable* strategy, there is exactly one token for the field, which contains the value of the field. The field can only be changed when owning the token. In the *count* strategy, there can be any number of tokens, and the value of the field is the sum of all tokens.

Rather than having a protected environment part of the state as in our approach, they use tokens to protect fields from changes. When constructing an atomic variable, they have to hand over the token of the field which

represents this variable. The token also replaces the `AtomicId` from our approach.

They can define inductive invariants, which must be preserved by all transitions. These are part of the definition of the state machine, rather than stating them inline in the implementation as in our approach.

It would be interesting to implement their producer-consumer queue example in our approach, in order to compare the two approaches. It should be possible to implement their `PCell`, a verified version of Rust's `UnsafeCell`, in our approach. To do so, we would need to remember whether the `PCell` is initialized in the abstract state, and check that subsequent accesses are related by happens-before to prevent data races.

Each instantiation of the state machine is identified by an instance, and tokens are associated with that instance. The instance and the initial set of tokens is returned by the same method call. This way, no assumptions about initialization are needed as in our approach (Section 3.7.1), as long as we only have concurrency but not a distributed system. Similarly, the fact that only a single action can be performed in a state machine step avoids the assumption that each ghost lock step terminates (Section 3.7.2). It might make sense to give up the ability to perform multiple actions in the same step in our approach, if it allows us to remove the termination assumption. We have not actually taken advantage of this ability in our case studies.

4.4 Limitations of Prusti

Here we discuss current limitations and missing features in Prusti, and how these could be addressed, based on our experience in this work. The implementation of most of these features is described by Maier [10], but they are not yet functional in the version of Prusti used for the evaluation of this work.

4.4.1 Abstract data types

It would be useful to have the abstract types `Int`, `Id`, `Set<T>`, `Map<K, V>` and `Seq<T>` as part of the Prusti standard library. Many of these already exist in Viper [19], the intermediate verification language which Prusti targets, so it should be possible to expose them in Prusti. `Int` is the unbounded integer type.

`Id` is an identifier without inherent meaning; it could be implemented as a Viper domain. This type is useful in combination with abstract functions, which allow us to attach meaning to the identifier. In our implementation, we used `u32` as the identifier type. This is not entirely correct, since it means that there exists only a finite set of identifiers; but in practice it is not an issue,

as the verification tool will not exploit this fact on its own. The identifier type needs to support defining constants; with `u32` we use arbitrary numbers to define the values of constants.

The set, map and sequence types can be implemented using `Id` and abstract functions, and we did so in our implementation. We also need axioms, e.g. to state that two set which contain the same values are identical. Prusti does not allow us to define axioms, but we can approximate it by stating the axiom as a postcondition on an abstract method or function. Our map type is total, meaning that it assigns a value to every key; if a partial map is needed, `Map<K, Option<V>>` can be used.

In Section 3.2.3, we described how we can create a map which maps each key to the value of an expression which depends on the key. This is cumbersome to use, as the expression has to be put in the postcondition of a separate function. But it serves as a proof of concept for a builtin map expression, which could look similar to existing quantifier expressions.

4.4.2 Ghost code and data

In our approach, we rely on ghost code and data; for example, the ghost lock is ghost data, and its methods are ghost code. However, Prusti does not yet support ghost code, which means that in our implementation, the ghost code ends up being executed at runtime. We can do this for a proof of concept, but this is not suitable for actual use.

The way this could work is that we could define methods to be ghost methods. These methods must be verified to terminate, and we could allow the extended specification expression syntax to be used in them. In regular methods, we could have ghost blocks, which contain ghost code and call ghost methods.

We also need ghost data, which can be stored in variables and struct fields, and passed between methods in parameters and return values. This ghost data can only be accessed by ghost code, and does not exist at runtime. One way to achieve this is to define a wrapper type `Ghost<T>`, which contains ghost data of type `T` [10, p. 22]. This type is like `Box<T>`, the heap-allocated pointer in Rust, except that its size is zero instead of the size of a pointer, and it can only be accessed in ghost code.

Guards in our approach must be unique, so it must not be possible to copy a guard. But in some situations, we would like to make ghost copies of a resource, so that we can later refer to the value it had at that point, even if the resource does not implement the `Copy` trait and thus cannot normally be copied. In Verus [18], there are actually two different types of ghost data: `spec` and `proof`. `spec` data can be copied arbitrarily, while `proof` respects ownership rules. There are two ways in which this idea could be

implemented in Prusti: Either we could have two different ghost wrapper types, similar to Verus. Alternatively, we could let the guard type itself be regular non-ghost data, but make all its fields ghost data by wrapping them in `Ghost<T>`. Then, the guard still exists during compilation, but is zero-size, such that it can be optimized away by the compiler. It would be possible to put a guard inside a `Ghost<T>` and thus make a copy of it, however, the methods of the guard cannot be used with this copy.

Chapter 5

Conclusion

We developed an approach for proving refinement between a TLA$^+$ model and a Rust implementation, and demonstrated it on two case studies. The main challenge was to develop the interface and specifications of the ghost lock and guard, in a way that is practical to use but also sound; we have collected these specifications in Appendix A. Abstract functions turned out to be a very useful tool for this, as they can be used almost like a relational database.

We have implemented several ideas that we have not seen in existing refinement approaches, such as reduction checking using a Boolean flag, or the ability to state preserved predicates inline in the implementation, as opposed to defining invariants as part of the state machine itself.

There is still work left to do for the approach to become practical for use in the software industry. We ran into several limitations and bugs in Prusti. The most severe limitation is the lack of support for ghost code. We have fixed a small number of limitations in the Prusti code base, but in most cases, we worked around them, by writing our code in a different way. These workarounds are marked with comments in our case studies, which may serve as a reference for future improvements to Prusti. However, after implementing all these workarounds, it is impressive what is already achievable today with Prusti.

The approach itself still makes some assumptions that are not checked automatically. We could either just accept this, or try to lift these assumptions, as discussed in Section 3.7. Verus Transition Systems [17], which we only learned about very recently, is quite similar to our approach in some ways, and could serve as inspiration here.

Our approach to modularity works well for the hash set case study, but to evaluate how suitable it is for the verification of large applications, a larger

case study would be useful. It also adds more assumptions that need to be checked manually.

Finally, we could increase confidence in the approach by formally verifying its soundness in a proof assistant. This boils down to proving that our specifications of trusted methods are sound.

Appendix A

Ghost lock and guard specifications

Here, we show all specifications of the ghost lock and guard.

struct `GhostLock` {
  locked: `bool`, *did_act*: `bool`, *id*: `Id`,
  pub *state*: `AbsState`, *env_state*: `EnvState`
}

struct `Guard` {*kind*: `GuardKind`, *last_id*: `Id`}

const $INIT\_ID$: `Id`

abstract function *old_state*(*s*: `AbsState`): `bool`

abstract function *released*(*id*: `Id`): `bool`

abstract function *initial_state*(*id*: `Id`): `AbsState`

abstract function $final\_state$(*id*: `Id`): `AbsState`

abstract function *guard_is_open*(*id*: `Id`, *kind*: `GuardKind`): `bool`

abstract function *guarded_twostate*(
  s: `AbsState`, *n*: `AbsState`, *k*: `GuardKind`): `bool`

abstract method `GhostLock::new`(*init_state*: `AbsState`)
  requires *init*(*init_state*)
  ensures $\neg$*result.locked*
  ensures *old_state*(*init_state*)
  ensures *released*($INIT\_ID$)
  ensures $final\_state(INIT\_ID) = init\_state$

abstract method `GhostLock::acquire`(**&mut** *self*)
  requires $\neg self.locked$
  ensures $self.locked$
  ensures $\neg self.did\_act$
  ensures $initial\_state(self.id) = self.state$
  ensures $self.env\_state = get\_env\_state(self.state)$

abstract method `GhostLock::release`(**&mut** *self*, *action*: `Action`)
  requires $self.locked$
  ensures $\neg self.locked$
  requires $self.env\_state = get\_env\_state(self.state)$
  requires $next(initial\_state(self.id),\ self.state,\ action)$
  requires $\forall\ kind.\ action\_requires\_guard(action,\ kind) \rightarrow$
    $guard\_is\_open(self.id,\ kind)$
  ensures $old\_state(\mathbf{old}(self.state))$
  ensures $released(\mathbf{old}(self.id))$
  ensures $final\_state(\mathbf{old}(self.id)) = \mathbf{old}(self.state)$

abstract method `GhostLock::release_stutter`(**&mut** *self*)
  requires $self.locked$
  ensures $\neg self.locked$
  requires $self.env\_state = get\_env\_state(self.state)$
  requires $initial\_state(self.id) = self.state$
  ensures $old\_state(\mathbf{old}(self.state))$
  ensures $released(\mathbf{old}(self.id))$
  ensures $final\_state(\mathbf{old}(self.id)) = \mathbf{old}(self.state)$

abstract method `GhostLock::preserved_predicate`(
    &*self*, *s*: `AbsState`, *predicate*: `AbsState` $\rightarrow$ `bool`)
  requires $self.locked$
  requires $\neg gl.did\_act$
  requires $old\_state(s)$
  requires $predicate(s)$
  requires $\forall s, n, a.\ predicate(s) \wedge next(s, n, a) \rightarrow predicate(n)$
  ensures $predicate(initial\_state(self.id))$

abstract method `guarded_preserved_predicate`(
    last_state: `AbsState`, *state*: `AbsState`,
    kind: `GuardKind`, *predicate*: `AbsState` $\rightarrow$ `bool`)
  requires $guarded\_twostate(last\_state,\ state,\ kind)$
  requires $predicate(last\_state)$
  requires $\forall s, n, a.\ predicate(s) \wedge next(s, n, a) \wedge$
    $\neg action\_requires\_guard(a,\ kind) \rightarrow predicate(n)$
  ensures $predicate(state)$

abstract method `Guard::new`(*kind*: `GuardKind`)
  ensures *result.kind* = *kind*
  ensures *result.last_id* = *INIT_ID*

abstract method `Guard::open`(**&mut** *self*, *gl*: **&**`GhostLock`)
  requires *gl.locked*
  requires ¬*gl.did_act*
  requires *released*(*self.last_id*)
  ensures *self.kind* = **old**(*self.kind*)
  ensures *self.last_id* = *gl.id*
  ensures *guard_is_open*(*gl.id*, *self.kind*)
  ensures *guarded_twostate*(
    final_state(**old**(*self.last_id*)), *initial_state*(*gl.id*), *self.kind*)

Bibliography

[1] A. Bílý, C. Matheja, and P. Müller, "Flexible refinement proofs in separation logic," 2021. https://arxiv.org/abs/2110.13559

[2] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification," in *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, vol. 3, no. OOPSLA. ACM, 2019, pp. 147:1–147:30. https://doi.org/10.1145/3360573

[3] N. D. Matsakis and F. S. Klock, "The Rust language," *Ada Lett.*, vol. 34, no. 3, p. 103–104, oct 2014. https://doi.org/10.1145/2692956.2663188

[4] L. Lamport, *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. https://lamport.azurewebsites.net/tla/book.html

[5] R. J. Lipton, "Reduction: A method of proving properties of parallel programs," *Commun. ACM*, vol. 18, no. 12, p. 717–721, dec 1975. https://doi.org/10.1145/361227.361234

[6] Prusti user guide. Accessed: 2023-05-01. https://viperproject.github.io/prusti-dev/user-guide/

[7] J.-H. Jourdan. Insufficient synchronization in arc::get_mut. Accessed: 2023-04-22. https://github.com/rust-lang/rust/issues/51780

[8] std::memory_order. Accessed: 2023-04-21. https://en.cppreference.com/w/cpp/atomic/memory_order

[9] O. Lahav, N. Giannarakis, and V. Vafeiadis, "Taming release-acquire consistency," *SIGPLAN Not.*, vol. 51, no. 1, p. 649–662, jan 2016. https://plv.mpi-sws.org/sra/

[10] J. Maier, "Towards verifying real-world Rust programs," Bachelor Thesis, ETH Zürich, 2022. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Jonas_Maier_BS_Thesis.pdf

[11] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, December 2001. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[12] L. Lamport. The Paxos algorithm or how to win a Turing Award. Accessed: 2023-04-27. https://lamport.azurewebsites.net/tla/paxos-algorithm.html

[13] M. A. Kuppe. OpenAddressing.tla. Accessed: 2023-04-29. https://github.com/tlaplus/tlaplus/blob/master/tlatools/org.lamport.tlatools/src/tlc2/tool/fp/OpenAddressing.tla

[14] VerifyThis competition 2022. Accessed: 2023-05-03. https://www.pm.inf.ethz.ch/research/verifythis/Archive/2022.html

[15] TLC model checker. Accessed: 2023-05-01. https://github.com/tlaplus/tlaplus

[16] C. Hawblitzel, J. Howell, M. Kapritsos, J. Lorch, B. Parno, J. Stephenson, S. Setty, and B. Zill, "IronFleet: Proving practical distributed systems correct," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM - Association for Computing Machinery, October 2015. https://www.microsoft.com/en-us/research/publication/ironfleet-proving-practical-distributed-systems-correct/

[17] T. Hance. Verus transition systems. Accessed: 2023-05-01. https://verus-lang.github.io/verus/state_machines/

[18] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying Rust programs using linear ghost types," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, apr 2023. https://doi.org/10.1145/3586037

[19] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62. https://pm.inf.ethz.ch/publications/MuellerSchwerhoffSummers16.pdf

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Proving Refinement in a Rust Verifier

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Schär	Jan

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 2023-05-10	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.