# Proving Refinement in a Rust Verifier

Master Thesis Project Description

Jan Schär
Supervised by Prof. Dr. Peter Müller, Aurel Bílý, João Pereira
Department of Computer Science
ETH Zürich
Zürich, Switzerland

## I. Introduction

Prusti [1] is a tool for the verification of Rust [2] 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$^+$ [3] 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 implement support in Prusti for describing and verifying refinement between a TLA$^+$ model and an implementation written in Rust, using the approach described in "Flexible Refinement Proofs in Separation Logic" by Bílý *et al.* [4]. We only target safety properties, liveness properties are out of scope.

## II. Approach

To summarize the approach by Bílý *et al.* [4]:

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 1 shows an example ATS, written in the TLA$^+$ language. Fig. 1 shows a trace of this ATS.

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

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

$$\begin{aligned} Next \triangleq\ & \land count' = count + 1 \\ & \land stdout' = Append(stdout, count) \end{aligned}$$

Listing 1. Example TLA$^+$ model. In *Next*, unprimed and primed variables refer to the state before and after the transition, respectively.

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 that, we need to describe the execution of the program in terms of the model variables.

$$\begin{tikzcd}
\boxed{\begin{array}{c} \mathit{count} = 0 \\ \mathit{stdout} = \langle\rangle \end{array}} \arrow[r] & \boxed{\begin{array}{c} \mathit{count} = 1 \\ \mathit{stdout} = \langle 0 \rangle \end{array}} \arrow[r] & \boxed{\begin{array}{c} \mathit{count} = 2 \\ \mathit{stdout} = \langle 0, 1 \rangle \end{array}}
\end{tikzcd}$$

Fig. 1. A possible trace (sequence of states) of the example.

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 functions 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` function, 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 functions 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, where the ghost lock is acquired at the start and released at the end: (1) The `Next` relation must hold for the model state before and after the block. (2) The code executed in the block can perform at most one atomic observable action. 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.

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.

## III. Implementation in Prusti

Our goal is to implement this approach in Prusti. To get an idea of what is required, let's again look at the example. Listing 2 shows a Rust program, which implements the example model. We have added ghost code in the syntax from Bílý *et al.* [4].

*a) Model:* We need to define how to write the TLA$^+$ model in Rust syntax. This includes a definition of the variables and their types, and the `Init` and `Next` relations. We may also need to define which variables are environment variables.

In general, the `Next` relation is a disjunction of different transitions, and each transition may have existential quantifiers. Automation can have difficulty determining which transition to use and how to instantiate the quantifiers. For this reason, we plan to annotate in the code which transition is taken and to provide witnesses for the quantifiers. To do this, the transitions and quantifiers need to have identifiers.

```
stdout := Append(stdout, x)
fn print(x: i32) {
  println!("{}", x);
}

fn main() {
  count := 0;  stdout := ⟨⟩
  Init {
    let mut c = 0;
    while c < 5 {
      Next {
        print(c);
        count := count + 1
      }
      c = c + 1;
    }
  }
}
```

Listing 2. The example implemented in Rust, with ghost code on shaded background

For defining the relations, we plan to use Prusti predicates. The identification of transitions and quantifiers could be done by defining a Rust `enum`, with a variant for each transition, and defining fields on the variants for the existential quantifiers. The `Next` predicate can then be written as a `match` expression over this enum. However, due to current limitations of Prusti, we may need to use a solution based on function calls instead.

*b) Environment functions:* The `print` function interacts with the environment, so it has a specification which describes this interaction on the model state. In this case, it appends the printed value to the *stdout* sequence.

This specification needs to be written as a Prusti annotation on the function.

*c) Ghost lock:* The 'Init' block initializes the ghost lock. It is written as a block to indicate that the ghost lock remains initialized during the execution of the program. The 'Next' block acquires the ghost lock. Only inside this block are ghost variables allowed to be modified.

We need to define the Rust syntax for initializing and acquiring the ghost lock. Probably, we will define Rust macros for this. We also need to define how to access and modify the model variables protected by the ghost lock.

*d) Atomicity:* We must check that each 'Next' block performs at most one atomic interaction with the environment, so that it can be accurately modeled as a single transition in the TLA$^+$ model. How to do this is to be investigated, it may involve implementing an analysis in Prusti.

*e) Regular locks:* We also want to add support to Prusti for verifying regular locks (i.e., `Mutex` in Rust). This is relevant for the verification of multithreaded programs. There are similarities to the ghost lock, so we may be able to use some mechanisms that we will implement for both.

We will need some way to attach invariants to locks. One possibility could be to do something similar to *specification entailments* from "Modular Specification and Verification of Closures in Rust" [5], but for locks instead of closures.

*f) Guards:* As mentioned in Section II, we need some way to keep knowledge about the model state while not holding the ghost lock. Consider the `print(c)` call in Listing 2. We need to prove that the concrete variable `c` has the same value as the model variable *count*. *Guards* are one possible solution to this, used in some of the examples of Bílý *et al.* [4].

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 unless a mutable reference to a particular guard is provided. For example, we can define a guard `CountGuard`, and only allow changing *count* if a `&mut CountGuard` is provided.

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 'remember' the value of these predicates, while not holding the lock. Because the thread owns the guard, and the guard is unique, no transitions which require the guard can happen elsewhere.

## IV. Core goals

1) Introduce a minimal and sound extension to the specification language of Prusti and its logic:
   - Enable reasoning about atomicity of a given MIR control-flow (sub)graph.
   - Implement an encoding for at least one physical lock, e.g. `std::sync::Mutex`.
   - Implement an encoding for at least one ghost lock, e.g. `GhostLock` attached to an abstract transition system + invariants.
   - Encode `init` and `next` steps.
2) Describe a canonical format for TLA$^+$ specifications that allows:
   - A clear identification of (two-state) invariants, as seen in the `Next` relation in our example.
   - A clear identification of transitions and related existentially quantified variables.
3) Demonstrate this approach in a Case Study implemented in Rust and verified with Prusti. Possible case studies include implementations of distributed hash table algorithms (like Chord), and consensus algorithms such as Paxos and Raft.

## V. Extension Goals

1) Add support for guards in Prusti.
2) Add support for interfacing Prusti with TLA$^+$ modules to allow importing TLA$^+$ models directly in Prusti.
3) Allow a clear separation of state between different components at the code level.
4) Support component-wise initialisation, i.e. `Init` state reached at different times in different threads.

## VI. Schedule

| | |
|---|---|
| Implement lock encoding | 3 weeks |
| Implement ghost lock | 3 weeks |
| Reason about atomicity | 3 weeks |
| Create case study | 4 weeks |
| Extension goals | 3 weeks |
| Write report | 4 weeks |

## References

[1] 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. [Online]. Available: https://doi.org/10.1145/3360573

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

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

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

[5] F. Wolff, A. Bílý, C. Matheja, P. Müller, and A. J. Summers, "Modular specification and verification of closures in Rust," in *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, vol. 5, no. OOPSLA. New York, NY, USA: ACM, oct 2021. [Online]. Available: https://doi.org/10.1145/3485522