Extending IDE Integration of a Rust Verifier

Practical Work Description

Joseph Thommes, Cedric Hegglin
Supervised by Prof. Dr. Peter Müller, Aurel Bílý

September 26, 2022

1 Introduction

Prusti [Ast+19] is a verifier for Rust using the Viper [Juh+14] framework for verification. It offers an IDE extension for VS Code [Mic22] that facilitates working with Prusti: Prusti Assistant. This extension already allows verifying files from within the IDE but lacks some more advanced features to make it more usable. This practical work aims to implement some of these features.

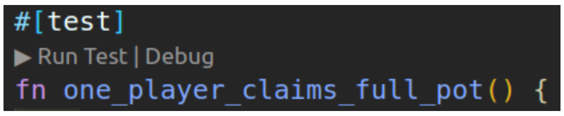
2 Goals

2.1 Flowistry-related goals

Many desirable features of Prusti Assistant involve some analysis of a given Rust program. Flowistry [Cri+22] is a modular information flow analysis tool for the Rust programming language. Apart from computing information flow data, Flowistry computes other data about the program such as a mapping of source code positions to their corresponding MIR encoding. This might greatly simplify the implementation of some useful features. The goal of this section is to start using Flowistry in Prusti, to then implement the features described in the rest of this section, using the newly available information.

Selective verification Verification of programs can be very time consuming, especially for large code bases. Oftentimes however, users only work on one or a few methods before retrying verification. Every invocation of Prusti via Prusti Assistant will verify all methods that are not already cached, even though the verification is modular in the background.

To expose the benefits of modular verification to the user, we want to extend Prusti to allow verifying subsets of methods at a time, and make this functionality available via Prusti Assistant. Ideally this would work



```
#[test]
▶ Run Test | Debug
fn one_player_claims_full_pot() {
```

Figure 1: Example of a Rust testcase using the VSCode extension rust-analyzer, providing a button to run this specific function.

in a fashion similar to test-interfaces provided my many IDEs, to aid user familiarity. An example of this functionality is shown in Figure 1, displaying how single tests can be run with the click of a button. In a similar fashion a verify button above each function could be very useful.

Simplify declaring external specifications In any real-world program there will be calls to methods that are not part of the crate we are currently verifying, e.g. standard library methods. In Prusti it is possible to create specifications for these methods, but this is quite tedious since it requires the user to find and duplicate the original method's declaration. An example of this is shown in Listing 1, where all the content below line 6 is required for the main method to verify. To simplify this, we plan to automate this process by allowing users to select an external method's call they wants to annotate. This in turn will find the method's signature and either inline it into the program, or generate a separate file where one can annotate the method with specifications.

```
1  fn main() {
2      let x = Some(5);
3      assert!(x.is_some());
4  }
5
6  #[extern_spec]
7  impl<T> std::option::Option<T> {
8      #[pure]
9      #[ensures(matches!(*self, Some(_)) == result)]
10     pub fn is_some(&self) -> bool;
11 }
```

Listing 1: Example where the external function `is_some` needs additional specifications for successful verification of the function `main`.

Map source code positions to Viper While most of the goals so far are more user oriented, the next one will mostly be helpful to developers working on Prusti itself and its translation to Viper. When Rust functions are translated to Viper, the generated files become very verbose and are hard to read. To debug the translation of Prusti however, one often has to find the corresponding Viper code for certain statements in Rust. Therefore a mapping of positions in the original Rust program to the generated Viper program would be very helpful.

The accuracy of this mapping is yet to be determined. Depending on the available information in Prusti the goal is to either just find the corresponding method in Viper or the corresponding basic block.

2.2 Quantifier-related challenges

This section deals with some open goals in the context of quantifiers.

Quantifiers (e.g. `forall`) in the Prusti specification are translated into Viper quantifiers, which in turn are translated into SMT-LIB quantifiers, checked by Z3 [MB08].

Quantifier mapping Unfortunately, there is no easily accessible mapping of the quantifiers a user enters in the program to the ones used in the backend by Viper and Z3. This poses a problem on its own as it can hinder debugging: the debug messages/errors from the lower layers in the verification pipeline are referring to the quantifiers on their respective layer – rendering an interpretation of these pieces of information on the front end layer difficult.

```
#[ensures(
    forall(|i: usize|
        (0 <= i && i < self.len() && i != index)
        ==> (self.lookup(i) == old(self.lookup(i)))
    )
)]
```

Listing 2: Example of a `forall` quantifier in Prusti. Taken from [dev22]

Quantifier instantiations In order to assist debugging, it would be useful to report statistics about Z3's quantifier instantiations back to the front end. This would allow e.g. the detection of unintended instantiation loops and in general allow for a more targeted debugging.

A necessity for the implementation of this feature is the quantifier mapping mentioned before as one needs to know the correspondence between the Z3 quantifiers in the backend and the Prusti ones in the front end.

Quantifier triggers Another related topic are the quantifier triggers in Prusti/Viper that are used to specify the "domain" of a quantifier. These can be stated manually by the user or be inferred automatically by Viper. In the latter case, a report on the choice of these triggers can facilitate debugging in Prusti as well as developing it.

2.3 Other goals

This section lists goals that do not fall into one of the earlier categories.

Per-method verification times Prusti Assistant only displays the total verification time, meaning the time taken to verify all methods of a file or crate. The time needed to verify a specific method can differ largely and oftentimes gives a good indication on where some unintended behavior might be happening. The goal here is to make per-method verification times visible in the IDE.

Cache awareness When verifying a program, Prusti will use cached results of a previous verification if a method hasn't changed. Prusti Assistant is not aware of this since it is all handled by the backend, but in combination with the per-method verification times it would be interesting for a user to know whether a result comes from cache or not. Therefore this should also be reported in the IDE.

Error differentiation Another shortcoming of Prusti Assistant is that it reports compilation and verification errors to the user in the same style, which makes them difficult to differentiate.

As the actions to be taken on a verification error are very different to the ones taken on a compilation error, it makes sense to also visually differentiate between the two kinds of errors (e.g. in the status bar).

Prusti snippets VS Code offers so-called snippets that allow for repeating code patterns to be auto-completed on writing. Only a subset of Prusti keywords/constructions are supported, which should be extended to include, for example, pledges.

Counterexample reporting Over the last two years several students, including one of us, worked on extending Prusti to produce counterexamples whenever verification fails. For people that use Prusti via command line these can already be quite helpful. For users of the VS Code extension however, it is quite tedious to activate them. One simple improvement would be to add an option in the settings to enable counterexamples. Also, at the moment these counterexamples are only displayed as error messages in the debug tab of VS Code. These error messages already contain the span of the variables in the program they refer to, which opens an opportunity to improve their presentation. The goal is to provide so-called inlay hints behind each variable, containing the value of that variable that apparently causes the program to fail verification. However, for large or unbounded data structures where counterexamples can contain a lot of information, this way of presenting them might not be suitable. The solutions to solve this problem are still in discussion.

Robust setup process Another improvement on the Prusti Assistant should be a more robust setup process. This process should check if the dependencies downloaded or are already installed, be able to handle interrupted downloads and handle any kind of error accordingly. Another idea would be to bundle versions of Prusti with Prusti Assistant to avoid the additional download step for the most basic use case.

3 Working Schedule

We plan on having weekly meetings in order to discuss the current progress and the challenges that arise during implementation.

This practical work is scheduled for six months and there is no further structure to its schedule apart from gradually implementing all possible features listed under section 2. However, the goals in 2.1 will be handled by Cedric Hegglin, whereas Joseph Thommes is responsible for the goals in 2.2. The remaining features to implement will be handled as time allows and may be done by either of the two students.

References

[MB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[Juh+14] Uri Juhasz et al. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: (2014). URL: `https://pm.inf.ethz.ch/publications/JKMNSS14.pdf`.

[Ast+19] V. Astrauskas et al. "Leveraging Rust Types for Modular Specification and Verification". In: *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).* Vol. 3. OOPSLA. ACM, 2019, 147:1–147:30. DOI: `10.1145/3360573`. URL: `http://doi.acm.org/10.1145/3360573`.

[Cri+22] Will Crichton et al. "Modular Information Flow through Ownership". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 1–14. ISBN: 9781450392655. DOI: `10.1145/3519939.3523445`. URL: `https://doi.org/10.1145/3519939.3523445`.

[dev22] Prusti developers. *Prusti User Guide.* 2022. URL: `https://viperproject.github.io/prusti-dev/user-guide/syntax.html` (visited on 09/28/2022).

[Mic22] Microsoft. *Visual Studio Code.* 2022. URL: `https://code.visualstudio.com/` (visited on 09/29/2022).