Counterexamples for a Rust Verifier

Bachelor's Thesis Description

Hegglin Cedric
Supervised by Dr. Christoph Matheja, Aurel Bílý
under Prof. Dr. Peter Müller

Version of October 23, 2020

1 Introduction

Prusti [1] is a verifier for the Rust programming language. It allows users to specify functional properties within Rust programs and can then prove whether these properties and all assertions hold. Internally, Prusti translates Rust programs to the Viper [2] intermediate language and uses the Silicon backend to do symbolic execution and check whether certain safety properties, assertions, and pre-/postconditions hold. While Silicon has the functionality to produce counterexamples when verification fails, Prusti does not yet process this information. The goal of this thesis is to enable the generation of *counterexamples* in Prusti and present them in a meaningful way to Rust developers.

Counterexamples in the context of formal verification tools are values for the program variables at various program points that explain how an error state can be reached. They are highly useful for debugging since they may allow a user to reproduce violations of specified properties. However, finding them is non-trivial and the limitations of verification also affect the generation of counterexamples. In particular, many tools may generate spurious counterexamples when they fail to prove a property that is not actually violated.

While some verification tools, such as Nagini [3] or Dafny [4] already support the generation of counterexamples, Prusti does not yet implement this functionality. To put this in context, consider, for example, a function `sum(x)` that computes the sum of all integers from one to `x`. This function also has a closed-form solution for positive arguments $\frac{x(x+1)}{2}$. We try to prove that `sum(x)` computes this solution with the mentioned verification tools, but we intentionally forget the precondition that `x` has to be non-negative, hoping to get a hint about this from our verification tools.

When running Prusti on a Rust implementation of this example (see Listing 1), it simply tells us that verification fails, which pre-/postcondition

```rust
use prusti_contracts::*;

#[ensures(result == x*(x+1)/2)]
fn sum(x:i32) -> i32 {
    if x <= 0 {
        x
    } else {
        x + sum(x-1)
    }
}
```

Listing 1: example program in Rust with Prusti annotations

or assertion might not hold and that the error results from the first if branch (Listing 2). In the following, we will compare this to the two previously mentioned tools, Dafny and Nagini, and look at their outputs for an analogous implementation of `sum(x)` in their supported programming language.

```
Verification of 2 items...
error: [Prusti: verification error] postcondition might not
    hold.
  --> pure.rs:3:11
   |
3  | #[ensures(result == x*(x+1)/2)]
   |           ^^^^^^^^^^^^^^^^^^^
   |
note: the error originates here
  --> pure.rs:4:1
   |
4  | / fn sum(x:i32) -> i32 {
5  | |     if x <= 0 {
6  | |         x
7  | |     } else {
8  | |         x + sum(x-1)
9  | |     }
10 | | }
   | |_^

Verification failed
error: aborting due to previous error; 1 warning emitted
```

Listing 2: Prusti's output when run on example

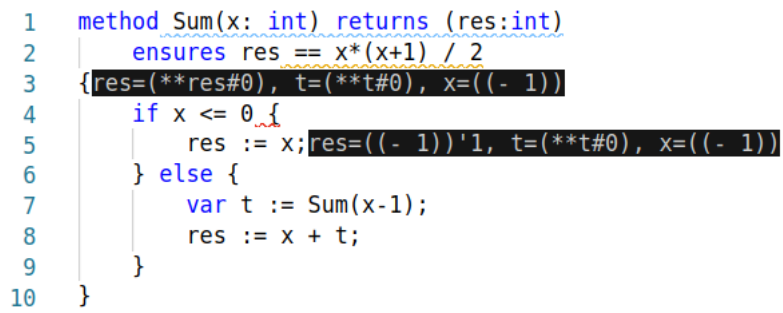

Figure 1: Dafny's counterexample within written code

```
Verification failed
Errors:
Postcondition of sum might not hold. Assertion (Result() == ((x
    * (x + 1)) // 2)) might not hold. (sum2.py@8.12).
Store:
  x -> -1
Heap: Empty.
```

Listing 3: Nagini's output when run on example

In Dafny with its Visual Studio Code extension, counterexamples are displayed within the written code showing the value of variables when entering the function and when returning from it. For the given example in Figure 1, the counterexample tells us that the property will be violated when entering the function with `x=-1` on line 3 and that the function will return `res=-1` in this case (line 5). Nagini outputs the input value that results in a violation and the current state of the heap (Listing 3). From both examples, it is easy to figure out that the function fails for negative inputs which was our goal. This shows that there is room for improvement in Prusti and also presents some options on how to present counterexamples.

2 Approach

Prusti is built on top of Viper which already implements some support for counterexamples. The goal is to leverage this functionality by backtranslating Silicon's generated counterexamples to the matching Rust variables for the Viper programs generated by Prusti.

Since the translation from Rust to the Viper intermediate language done by Prusti creates much more complex Viper programs, the counterexamples are also harder to understand than the previous ones from other tools. The counterexample that is generated by Silicon when letting Prusti translate the program in Listing 1 to Viper (see Listing 4) has a lot more variables, but the essential information that the program fails for negative inputs is still conserved in variable _3 with value -1. For this simple example, our algo-

```
Silicon found 1 error in 4.04s:
  [0] Assert might fail. Assertion (unfolding acc(i32(_0),
    write) in _0.val_int) == old[pre]((unfolding acc(i32(_1),
    write) in _1.val_int)) * (old[pre]((unfolding acc(i32(_1),
    write) in _1.val_int)) + 1) / 2 might not hold. (pure.rs.
    vpr@354.3)
__t9 -> false
__t10 -> false
_1 -> $Ref!val!0
_4 -> 0
_5 -> $Ref!val!3
_6 -> $Ref!val!3
_7 -> 0
_8 -> $Ref!val!3
_9 -> $Ref!val!3
_3 -> (- 1)
_2 -> $Ref!val!2
__t8 -> true
_0 -> $Ref!val!3
```

Listing 4: Silicon's output when ran on Prusti-generated code from previous example

rithm should be able to map the variable `_3` from the Viper program back to `x` in our original Rust program. Extracting this information automatically, even for more complex programs, is the main goal of this project.

Another important point is user-friendliness. Compared to other verification tools, Prusti tends to require fewer user annotations as, thanks to Rust's strong type system, they can often be inferred automatically. Integrating counterexamples into Prusti should comply with this design. In particular, the counterexamples should be understandable by Rust programmers without knowledge of Viper or a strong background in verification. Therefore part of the effort will go into making the counterexamples easily readable and making them available via the existing Visual Studio Code Extension of Prusti.

2.1 Core Goals

- Identify examples where generating counterexamples in Rust through backtranslation seems feasible. These examples will also serve as benchmarks for evaluating the implementation.
- Discuss how the desired backtranslated counterexamples should look like and design how they should be presented to the user.
- Design an algorithm that, given a Rust program that fails verification and produces a counterexample in the backend, can process this counterexample and output what the corresponding program variables in

Rust are. This algorithm should support programs processing the following types: primitive types (booleans, integers, chars), tuples, finite non-recursive enumerations (including structs), and references.

- Implement the designed algorithm for counterexample generation in Prusti.
- Evaluate the quality of the implementation based on the previously collected set of examples.
- Extend Prusti-Assistant, the Visual Studio Code Extension, to also display counterexamples in the previously discussed format to the user without having to use the command-line.

2.2 Extension Goals

- Extend the designed algorithm and implementation to support a larger set of types, such as boxes and recursive enumerations.
- When verification fails and Prusti outputs a counterexample, automatically create a unit test to simplify reproducing and understanding the violation or recognize spurious counterexamples.
- When Prusti finds a counterexample for a function, extend the precondition of this function to a conjunction of the existing precondition and "the negation of the counterexample" and retry verification. By doing this and possibly repeating it a few times one might be able to distinguish missed corner cases in the precondition from more meaningful violations.

3 Working Schedule

Analyze Prusti and Viper, find examples, and design algorithm.	4 Weeks
Extend Prusti with the described algorithm.	5 Weeks
Extend Visual Studio Code Extension.	1-2 Weeks
Write thesis and prepare presentation.	6 Weeks

References

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging rust types for modular specification and verification. *Proc. ACM Program. Lang.*, 3(OOPSLA):147:1–147:30, 2019.

[2] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[3] M. Eilers and P. Müller. Nagini: A static verifier for python. In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, pages 596–603. Springer, 2018.

[4] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.