Counterexamples for Complex Data Structures for a Rust Verifier

Bachelor's Thesis Project Description

Markus Limbeck
Supervised by Vytautas Astrauskas, Aurel Bílý
under Prof. Dr. Peter Müller
Department of Computer Science
ETH Zürich
Zürich, Switzerland

I. Introduction

Rust [1] is a general-purpose programming language which guarantees rich memory safety properties, e.g. well-typed Rust programs do not exhibit unexpected side effects through aliased references. Prusti [2] is a verification tool which uses this type system to simplify the specification and verification of Rust programs. Prusti is based on the Viper verification infrastructure [3]. Viper was developed at ETH Zurich and provides an architecture on which new verification tools can be developed simply and quickly. While the verification process constructs formal proofs in a sophisticated logic, most of these details are hidden behind a layer of abstraction. This allows a user of Prusti to write functional properties, e.g. assertions, pre-/postconditions, within the source code with a syntax similar to that of Rust program expressions.

If a verification fails, Prusti has some support for counterexamples. A counterexample is a list of assigned variables that demonstrate why a specific property does not hold. This is very helpful in detecting the root cause of an error. Therefore counterexamples have many advantages and are always desired in any verification tool.

While Prusti is often able to generate counterexamples for primitive data types [4], it struggles with more complex data structures. Therefore the main goal of this thesis is to extend the support of counterexamples for more advance data types, e.g. recursive enums. Even simple programs often need these complex data structures. Hence, it will improve Prusti's ability to generate counterexamples for many more programs and, as a result, it will help users identifying the root cause of their error.

II. Approach

To extend the support of counterexamples for complex data structures we need to find the current limiting factors for Prusti. This is best explained by an example. Listing 1 demonstrates how a *Linked List* can be implemented in Rust.

```
struct LinkedList {
    val: i32,
    next: Option<Box<LinkedList>>
}
```

Listing 1: implementation of a Linked List in Rust

Rust needs to know the amount of space a type takes up at compile time. To define a recursive type one needs to use an indirection such as a *box*. Then the space is known statically and the recursive type can be compiled. A box is a smart pointer which points to a heap allocated value of a generic type, in this case LinkedList. Listing 1 shows that the box type is essential for complex data structures. Prusti cannot generate a counterexample for a box type as listing 2 and listing 3 demonstrate:

```
fn box_panic(x: i32) {
    let y = Box::new(x);
    if *y > 0 {
        panic!();
    }
}
```

Listing 2: example with box type

```
note: counterexample for "x"
 initial value: 1
 final value:   1
note: counterexample for "y"
 final value:   Box {
   0: Unique {
       pointer: ?,
       _marker: ?,
   },
   1: Global,
}
```

Listing 3: counterexample for box type

While Prusti is able to generate a counterexample for the variable x, in this case 1, it is not able to find a

counterexample for the variable y. Since the box type is essential for complex data structures, it is a good starting point to extend Prusti for this type.

Another limiting factor for Prusti's counterexample generation is the available information generated by the backend Silicon [5]. Prusti works the following way: Prusti translates the Rust program into a Viper program which is then verified by Silicon. If verification fails, Silicon generates a counterexample which Prusti translates back to Rust. If Silicon does not provide enough information, it is not possible for Prusti to generate a counterexample without making changes to Silicon. Nevertheless, in many cases Prusti has all the information needed but is not able to make use of it. Listing 4 demonstrates what is meant by this.

```
#[ensures(result)]
fn sorted(list: LinkedList) -> bool { ... }
```

Listing 4: sorted function for Linked List

This function checks if a Linked List is sorted in ascending order. Of course the postcondition will not hold if the parameter is not sorted. Prusti is not able to generate a counterexample even though Silicon provides enough information for a counterexample. Listing 5 is a simplified output of Silicon's counterexample.

```
_1 <- Ref (\$Ref!val!0) {
  f\$next(perm: 1/1) <- Ref (\$Ref!val!2) {
    f\$val(perm: 1/1) <- Ref (\$Ref!val!3) {
      val_int(perm: 1/1) <- -2147479462
    }
  }
  f\$val(perm: 1/1) <- Ref (\$Ref!val!2) {
    f\$val(perm: 1/1) <- Ref (\$Ref!val!3) {
      val_int(perm: 1/1) <- -2147479462
    }
    val_int(perm: 1/1) <- 1
  }
}
_5 <- 1
_6 <- -2147479462
```

Listing 5: counterexample for Linked List from Silicon

In this case *_1* corresponds to the Linked List given by the parameter and *_5* and *_6* correspond to the value stored in the first and second element in that Linked List. In other words, the counterexample in Prusti would be a Linked List with two elements with the values 1 and -2147479462.

Once a counterexample has been generated, it raises the question on how to present it to the user. An unwise choice could lead to hard-to-read results which might hinder users more than help them. One possible way could be using the type definition. This is how Prusti currently handles primitive data types. Another way could be to decouple the references which would lead to a more structured result. Listing 6 shows what is meant by this.

```
note: counterexample for "list"
    initial value: LinkedList {
        val: 1,
        next: std::option::Option::Some(box1)
    },
    box1: Box {
        0: Unique {
            pointer: list1,
            _marker: ?,
        },
        1: Global,
    },
    list1: LinkedList {
        val: -2147479462,
        next: std::option::Option::Some(?)
    }
```

Listing 6: sorted function for Linked List

This prevents deeply nested structures, and especially for bigger data structures it improves readability dramatically. Additionally, Prusti could provide a special annotation for the type definition such that the user can specify how the output should look like, similar to the *formatted print* in Rust.

```
#[c_print("[{}] -> {}", val, next)]
struct LinkedList { ... }
```

Listing 7: annotated type

```
note: counterexample for "list"
    initial value: LinkedList {
      [1] -> [-2147479462] -> ?
    },
```

Listing 8: counterexample with annotated type

A. Core Goals

- Identify the current limitations of Prusti's counterexample generation, e.g. the box type, and find examples with complex data structures where Silicon provides enough information.
- Design an extension of the current algorithm such that it supports complex data structures and discuss how counterexamples should be presented.
- Implement the designed algorithm in Prusti to improve its counterexample generation.
- Test the implementation against the previously collected examples and evaluate its quality.

B. Extension Goals

- **More fine-grained counterexamples:** At the moment Prusti's counterexamples include only

values of variables at the beginning and at the end of a function. In addition to this, it would be beneficial for a user to analyze the changes for each variable throughout a function and to provide a more detailed counterexample in form of a control flow graph.

- **Counterexamples for models:** Prusti has a feature called models. A model is an abstraction over a type which can be used in specifications. For example the type *std::vec::Vec* has a very complicated internal representation but the only interesting part in respect to counterexamples are the contents stored by it. As listing 9 shows, it would be very helpful to have a counterexample for the model to identify why a specific specification does not hold.

```
#[model]
struct std::vec::Vec<T> {
    contents: GhostSeq<T>,
}
...
#[requires(vec.model().contents == [1, 2])]
fn foo(vec: Vec<i32>) { ... }
...
foo(vec![3, 4]); // ERROR: vec should
                 // be [1, 2]
```

Listing 9: Example for a model in Prusti

- **Improve Silicon's counterexample generation:** As already mentioned Prusti depends on the information provided by Silicon. While in many cases this is enough, some shortcomings of Silicon might be discovered during the work for this thesis. Fixing these issues in Silicon would further improve Prusti's counterexample generation.

- **Improve hard-to-read counterexamples:** Automatically generated counterexample often have strange values, like *_6* in listing 5. A desirable goal would be to improve the readability of a counterexample. This can be done by automatically adding additional Prusti specifications to the Rust program if the verification fails. Alternatively, there might be an option for the Z3 Solver [6] (the SMT Solver behind Viper) to specify which values should be preferred for counterexamples.

- **Identify spurious counterexamples:** In general, program verifiers have to be conservative and over-approximate the possible set of counterexamples. In the worst case this leads to spurious counterexamples. The user might have difficulties to distinguish between a real one and a spurious one. By adding additional specifications to the Rust program, Prusti can detect if a counterexample might be spurious and can inform the user accordingly.

III. Working Schedule

Task	Time
Identify the limitations of Prusti's current algorithm	2 weeks
Design the extension to support complex data structures	2 weeks
Implement the extension into Prusti	3 weeks
Evaluate the quality of the new algorithm	1 week
Extension goals	8 weeks
Write final report	4 weeks

References

[1] Rust programming language. Accessed on 2022-03-15. [Online]. Available: https://www.rust-lang.org/

[2] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification," ETH Zurich, Tech. Rep., 2019.

[3] 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.

[4] C. Hegglin, "Counterexamples for a rust verifier," Bachelor's Thesis, ETH Zurich, 2021.

[5] M. Schwerhoff, "Advancing automated, permission-based program verification using symbolic execution," Ph.D. dissertation, ETH Zurich, 2016.

[6] Microsoft. Z3 prover. Accessed on 2022-03-15. [Online]. Available: https://github.com/Z3Prover/