Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Counterexamples for Complex Data Structures for a Rust Verifier

Bachelor Thesis

Markus Limbeck

August 14, 2022

Advisors: Prof. Dr. Peter Müller, Vytautas Astrauskas, Aurel Bílý

Department of Computer Science, ETH Zürich

Abstract

Prusti is a deductive verification tool for the programming language Rust. It is built on top of the Viper verification infrastructure and allows users to write specifications directly in the source code and prove their correctness. In limited cases a counterexample is automatically generated for a verification failure. Counterexamples provide essential information to speed up the debugging process.

This thesis aims to improve the current counterexample support in Prusti. This is done by using a different Viper encoding and taking advantage of recent changes in Viper's backend Silicon. Specifically, a new algorithm is implemented to translate a counterexample of Silicon back to a suitable representation in Rust which can be reported back to the user. This approach is not only capable of generating counterexamples for more complex programs, it also provides more detailed counterexamples than the old approach.

Acknowledgements

First of all, I would like to thank my supervisors, Vytautas Astrauskas and Aurel Bílý, for their exceptional support during our weekly meetings and our countless Zulip conversations. Whenever I needed help, they always provided valuable advice.

I also want to thank my mother, Sonja Limbeck, for proofreading my thesis.

Finally I would like extend my special thanks to Prof. Dr. Peter Müller and the Programming Methodology Group for giving me the opportunity to contribute to their projects and research.

Contents

Chapter 1

Introduction

Individuals and businesses throughout the world are using software on a daily basis. They often entrust their software with critical tasks, and a faulty implementation can cause serious problems. As a consequence ensuring correctness, although very difficult, is an important goal in software development.

The general-purpose programming language Rust [6] can help achieving this goal. It guarantees rich memory safety properties, e.g. well-typed Rust programs do not exhibit unexpected side effects through aliased references. This prevents a variety of errors commonly present in languages like C/C++ [1].

Further guarantees for Rust programs can be provided by external tools. A deductive verification tool like Prusti [9] can be used to provide functional correctness to Rust programs. Prusti is based on the Viper verification infrastructure [12] developed at ETH Zurich. Functional properties like pre-/postconditions are written directly into the source code, while the sophisticated logic for formal proofs is hidden.

Listing 1.1 shows a function annotated with pre-/postconditions. Verifica-

```rust
struct X {
    val: i32,
}

#[requires(x.val == 0)]
#[ensures(result)]
fn always_equal(x: X, y: X) -> bool {
    x.val == y.val
}
```

Listing 1.1: Prusti annotations.

```
error: [Prusti: verification error] postcondition might not hold.
  --> test.rs:2:11
   |
 6 | #[ensures(result)]
   |           ^^^^^^
   |
note: counterexample for "x"
        initial value: X { val: 0, }
        final value:   X { val: 0, }
  --> test.rs:8:17
note: counterexample for "y"
        initial value: X { val: 1, }
        final value:   X { val: 1, }
  --> test.rs:8:25
note: counterexample for "result"
        final value:   false
  --> test.rs:9:1
```

Listing 1.2: Counterexample for always_equal.

tion of this function by Prusti will fail and will report that the postcondition might not hold. Rather than identifying the reason manually, Prusti can be configured to generate a counterexample.

The counterexample seen in Listing 1.2 provides variable assignments after the verification error. For example, the variable `x` has an initial value `X{val: 0,}` and a final value `X{val: 0,}`. If `always_equal(x, y)` would be executed with `X{val: 0,}` (initial value) as its first argument, `x` would hold the value `X{val: 0,}` (final value) at the point of verification failure. The second parameter `y` is analogous. The variable `result` corresponds to the return value of the function. It only contains a final value, in this case `false`. Using this information, it is easy to see why the precondition is not strong enough to guarantee the postcondition. Adding a stronger precondition, e.g. `#[requires(x.val == 0 && y.val == 0)]`, results in a successfully verifiable function.

Unfortunately, this counterexample generation has its limitations. If data types increase in complexity, Prusti often fails to produce a counterexample.

Listing 1.3 shows an implementation of a LinkedList and the same function as before but with different parameter types. A verification of this changed function will obviously still fail. Unfortunately, Prusti will not report a counterexample back. Its counterexample generation algorithm cannot deal with recursive data structures and, as a result, will fail to produce one.

The main goal of this thesis is to improve Prusti's support for counterexam-

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

#[requires(x.val == 0)]
#[ensures(result)]
fn always_equal(x: LinkedList, y: LinkedList) -> bool { ... }
```

Listing 1.3: Implementation of a LinkedList.

ples with the focus on complex data structures. This will be achieved by using a different type encoding.

At the time of writing Prusti is being refactored. As a result the counterexample support has to be changed as well. We will focus more on improving the counterexample algorithm for the refactored version than for the old version.

Since more complex data structures naturally result in more complex counterexamples, part of the effort is providing an improved way of presenting counterexamples back to users.

The main contributions of this project were:

- We improved Prusti's counterexample generation by using a different type encoding.
- We provided a new customizable way to present a counterexample.
- We adapt Prusti's counterexample generation algorithm to make it compatible with the refactored version.
- We discussed current limitations and how they could be resolved in the future.

In the following chapter we provide necessary information about Prusti and how it interacts with the Viper verification infrastructure. In Chapter 3 we give a high-level overview of our approach before we dive into the details in Chapter 4. In Chapter 5 we evaluate our implementation and compare it with Prusti's current one. Chapter 6 summarizes our project and describes potential future improvements.

Chapter 2

Background

In this chapter we describe the technical details needed for this thesis. We start by looking at Prusti's verification pipeline. In Section 2.2 we explain what a type encoding is and how it is used by Prusti. Section 2.3 describes relevant changes between the old and the refactored version. In the last section we introduce a Prusti feature called *model*.

2.1 Verification pipeline

In this section we will explain the verification process of Prusti on a very high level. In general, the program under verification is translated into the Viper intermediate verification language. Details about this language can be found in the Viper tutorial [8]. We assume at this point that the reader is familiar with the basics of this language. Once a program is translated, it can be verified by Silicon [13]. Silicon is one of Viper's backends and is based on symbolic execution. In the case of a failed verification, Silicon obtains a counterexample from the underlying SMT solver, Z3 [11]. This counterexample is translated back to the level of Rust and presented to the user in the form of additional information attached to the error message.

2.1.1 Translation

Here we will explain the translation process from Rust to the Viper intermediate verification language in more detail. While the whole translation is very complicated, the parts relevant for counterexamples are much more understandable.

Prusti is an extension of the Rust compiler and as such it has access to the compiler's internal information. The compiler transforms Rust code into various intermediate representations. One of these is called the mid-level intermediate representation (MIR) [3]. Prusti's translation is heavily based on the information given by this representation. Therefore we want to inspect it a bit more closely.

The MIR is based on a control-flow graph. It is made up of basic blocks which are uniquely identified by labels. All variables in the MIR, e.g. function arguments, have explicit types and are identified by an index written with a leading underscore, like _1. The variable name _0 refers to the return value of a function and the variable _i corresponds to the i-th argument of the function.

Prusti uses all of this information to generate a Viper program. Let us see how the Rust function from Listing 1.1 would look like in Viper.

```
method always_equal() returns (_0: Ref)
{
  var _1: Ref
  var _2: Ref
  var __t0: Bool
  var __t1: Bool
  var __t2: Bool
  var __t3: Bool
  label start
  __t0 := true
  ... // Preconditions
  label pre
  __t1 := true
  label l0
  __t2 := true
  _0.val_bool := _1.f_a.val_int == _2.f_a.val_int
  label l1
  __t3 := true
  assert _0.val_bool //Postcondition
}
```

Listing 2.1: We can see how the structure of the Viper program is influenced by the MIR. The variables _0, _1, _2 correspond to the equally named ones in the MIR. Therefore we know that _0 refers to the result value and _1, _2 to the arguments of the original Rust function. The other four variables indicate if a label, which refers to a basic block of the MIR, has been visited. The first label and its code block deal with preconditions and the last one deals with postconditions. The code in between encodes the actual functionality of the Rust function.

While Listing 2.1 demonstrates the idea of this translation process, it is only a simplified version of the actual Viper program. We have changed and shortened some names and omitted irrelevant information to improve readability. The key elements to understand here are that basic blocks in the MIR correspond to the code blocks after the labels and that each MIR variable has

```
__t0 <- true
__t1 <- true
__t2 <- true
__t3 <- true
_0 <- Ref (Ref!val!0) {
    val_bool(perm: 1/1) <- false
}
_1 <- Ref (Ref!val!1) {
    f_a(perm: 1/1) <- Ref (Ref!val!3) {
        val_int(perm: 1/1) <- 0
    }
}
_2 <- Ref (Ref!val!2) {
    f_a(perm: 1/1) <- Ref (Ref!val!4) {
        val_int(perm: 1/1) <- 1
    }
}
```

Listing 2.2: In Silicon's counterexample all variables are assigned some values. Primitive variables, e.g. __t0, are directly assigned a value, in this case true. References, like _0, contain fields which are assigned some values, in this case val_bool and false.

an equally named Viper variable.

2.1.2 Back translation

Once a Rust program is translated, it can be verified by Silicon. If the verification fails, Silicon provides a counterexample. The Viper program from Listing 2.1 would result in the following counterexample:

The counterexample from Listing 2.2 has to be translated back to Rust. This involves two steps. First, we filter out irrelevant variables. Second, we extract the values from Silicon's counterexample.

As already mentioned Viper variables have their equally named MIR counterparts. In addition, the compiler knows the relationship between MIR variables and the original Rust Variables. We can use this knowledge to collect tuples of Rust and Viper variables, e.g. (_0, return value), (_1, x) and (_2, y). We see that __t0, __t1, __t2, __t3 are not part of the final counterexample as they do not refer to any Rust variables. Extracting all the information from Silicon's counterexample results in the counterexample seen in Listing 1.2.

2.2 Type encodings

In the translation process all Rust types have to be encoded in Viper. This can be done via two different methods. The *heap-based type encoding* [2] uses Viper's predicates and references, while the *snapshot-based type encoding* [7] uses Viper's domains. We assume the reader is familiar with Viper's predicates and domains.

For the Section 2.2.1 and the Section 2.2.2 we assume we have a Rust variable `x` of type `struct X { val: i32 }`. We will explain how Prusti encodes the type using the two different methods. While the following examples showcase the idea of this translation process, the actual names in the real encoding are slightly different.

2.2.1 Heap-based type encoding

The heap-based type encoding produces a new predicate for each type which holds permissions for all of its fields. The encoding for our variable x looks the following way:

```
field X_a: Int
predicate X(self:Ref) {
  acc(self.X_a, write)
}
```

```
method fail(x: Ref)
    requires X(x)
{
    unfold X(x)
    assert (x.X_a != 0)
    fold X(x)
}
```

Listing 2.3: Failing method using a heap-based encoded type.

```
x <- Ref(ref!val!0) {
    X_a(perm: 1/1) <- 0
}
```

Listing 2.4: Counterexample for a heap-based encoded type.

Silicon will fail to verify the method from Listing 2.3 and will produce the counterexample shown in Listing 2.4. This counterexample looks similar to what we have previously seen, and Prusti is fully capable of translating it back to Rust.

2.2.2 Snapshot-based type encoding

The snapshot-based type encoding of the old version of Prusti and that of the refactored one only differ slightly. In the following section we only explain the newer encoding since most of the work of this project focuses on the refactored version. Detailed information about the old one can be obtained in the Prusti Developer Guide [7].

```
domain Snap_X {
  function constructor_Snap_X(f_a: Int): Snap_X
  function destructor_Snap_X_f_a(value: Snap_X): Int
  ...
  axiom ...
}
```

Listing 2.5: Snapshot-based encoding of struct X.

```
method fail(val: Int)
{
    var x: Snap_X
    x := constructor_Snap_X(val)
    assert (destructor_Snap_X_f_a(x) != 0)
}
```

Listing 2.6: Failing method using a snapshot-based encoded type.

```
constructor_Snap_X(Int) : Snap_X {
    0 -> x
    else -> #unspecified
}
destructor_Snap_X(Snap_X) : Int {
    x -> 0
    else -> #unspecified
}
```

Listing 2.7: Counterexample for a snapshot-based encoded type.

The snapshot-based type encoding uses domains instead of predicates. Each domain contains multiple functions and axioms. Since domain functions cannot have a body, axioms are needed to provide meaning to them.

Listing 2.5 shows the parts of the type encoding which are relevant for us. While the complete encoding contains more functions, we are mostly interested in *constructor* and *destructor* functions. Each field in the original type has a destructor function. Given a variable with this domain as its type, we can use them to "destruct" the variable to its field. The constructor does, as the name suggests, the opposite. Given variables for each field, we can "construct" the domain variable.

Again we use Silicon to verify the method seen in Listing 2.6. This will, of course, fail and will produce the counterexample shown in Listing 2.7. This counterexample looks vastly different to the one previously seen in Listing 2.4. The counterexample is now function-based rather than variable-based.

A function has entries in the form of input-output pairs and exactly one *default* case. We can think about these entries in the following way: Given some input for a function, the return value of a function is either the output of an entry or the default case. More details about this are found in Chapter 4.

This highlights why Prusti is not able to generate a counterexample anymore and why a new algorithm is needed. Fulfilling this need is the main goal of this thesis.

2.2.3 Heap-dependent vs heap-independent

We have now seen that the two encodings result in syntactically different counterexamples but we also want to emphasize that they might also be different semantically. This is due to the fact that one encoding depends on the heap and the other does not.

The heap-based type encoding is, as the name suggests, heap-dependent. In Viper, heap-dependent variables can only be accessed if the necessary permissions are held. We can think of variables whose permissions are not held at a certain program state as irrelevant at that program state, assuming permissions are correct. A counterexample produced by Silicon will only contain relevant variables. This means, depending on the program state of the verification error, a counterexample might be less precise than it would have been at an earlier program state because of missing permissions. More about this problem can be found in the previous work on counterexamples [10].

The snapshot-based type encoding is heap-independent. A counterexample for a domain function at a certain program state will be at least as precise as a counterexample at another earlier program state. This solves the problem and results in a more detailed counterexample in many cases.

In Listing 2.8 we see an example of the previously described problem. At the point of verification failure, line `17`, the permissions for the field `X_a` are not held anymore and therefore Silicon does not provide a counterexample for `x2`. For the function `destructor(x1)` Silicon does provide a counterexample. This shows that the heap-independence of the snapshot type encoding can result in more precise counterexamples.

2.3 Refactored Prusti

In this section we cover some important changes in the refactored version which are relevant to counterexamples.

First, the old version uses the snapshot-based type encoding only on demand, e.g. for checking structural equality, and therefore limits the places where the new counterexample algorithm can be used. In the refactored

```
1  field X_a: Int
2  predicate X(self:Ref) {
3     acc(self.X_a, write)
4  }
5  domain Snap_X {
6      function destructor(self: Snap_X) : Int
7  }
8
9  method fail(x1: Snap_X, x2: Ref)
10 requires X(x2)
11 {
12     var tmp: Int
13     unfold X(x2)
14     tmp := x2.X_a + 1
15     tmp := destructor(x1) + 1
16     fold X(x2)
17     assert(false)
18 }
```

Listing 2.8: The method `fail()` has two parameters. The first one has a domain type while the second one is a reference. A verification of the method will always fail and does not depend on the parameters. Line 13 and line 16 gain and release permissions for `x2` respectively.

```
fn some_method(mut x: i32) {
    x = 1;
    x = 2;
}
```

Listing 2.9: Rust function with a mutable parameter.

version every type is encoded via the snapshot-based type encoding and the new algorithm can always be used.

Another major change is the use of static single assignment (SSA). While in the old version Rust variables and Viper variables had a one-to-one relation, in the refactored version Rust variables and Viper variables have a one-to-many relation. Let us consider a simple example.

In Listing 2.9 we see a simple function which changes the value of its parameter. Furthermore, we assume that the MIR variable name of this variable is _1. In the old encoding, seen in Listing 2.10, a variables is assigned multiple times, e.g. in line 12 and line 17. In contrast, the refactored version, seen in Listing 2.11, creates a new variable for each assignment. The MIR variable _1 corresponds to the set {*_1_snap_0*, *_1_snap_1*} of Viper variables.

```
field value: Int
predicate Ints(self:Ref) {
    acc(self.value, write)
}
method some_method()
{
    var _1 : Ref
    inhale Ints(_1)
    ...
    label l1
    unfold Ints(_1)
    _1.value := 1
    fold Ints(_1)
    ...
    label l2
    unfold Ints(_1)
    _1.value := 2
    fold Ints(_1)
    ...
}
```

Listing 2.10: In the old version of Prusti variables like _1 are assigned multiple times. This is done by gaining permissions for the value field, assigning a new value to it and releasing its permission.

```
domain Snap_Ints {
    function constructor
    (self: Int) : Snap_Ints
    ...
}
method some_method()
{
    var _1_snap_0 : Snap_Ints
    var _1_snap_1 : Snap_Ints
    ...
    label l1
    inhale _1_snap_0
            == constructor(1)
    ...
    label l2
    inhale _1_snap_1
            == constructor(2)
    ...
}
```

Listing 2.11: The refactored version of Prusti uses SSA. Each variable is assigned exactly once. This is done via the inhale keyword and the constructor() function of the snapshot domain.

This poses new challenges, e.g. how to map Viper variables back to Rust variables for counterexamples, but it also provides new opportunities such as more fined-grained counterexamples. This is possible because we now have intermediate values stored in those Viper variables.

2.4 Type models

Structs might contain fields that are currently not supported by Prusti. A simple example would be `struct X { values: Vec<i32>}`. Fortunately, in this case Prusti supports a more straightforward mathematical representation: sequence.

The feature *type models* allows us to abstract our unsupported field via the supported mathematical type. A more detailed explanation of this feature can be found in Prusti's User Guide [4].

As we can see in Listing 2.12 the unsupported field in the original type is changed to its mathematical representation. Listing 2.13 demonstrates

```
#[model]
struct X {
    values: Seq<i32>,
}
```

Listing 2.12: A model is created by annotated a type via `#[model]`.

```
#[requires(x.model().values == ...)]
fn some_method(x: X) {
    ...
}
```

Listing 2.13: A model can be accessed in the specifications via `model()`.

how the model is used in specifications. This bypassing of unsupported features is very helpful, especially in the refactored version, for complex data structures.

Chapter 3

Approach

In this chapter we give an overview of how the snapshot-based type encoding is used to improve Prusti's counterexample generation. On a high level, we can split the changes into three parts: First, described in Section 3.1, we find a new mapping from Rust variables to Viper variables with domain types. From this point forward we will call these variables *snapshot variables*. Second, explained in Section 3.2, we extract the information from Silicon's counterexample and translate it back to the level of Rust. In Section 3.3 we describe how we provide the counterexample in a suitable way to the user.

3.1 Variable Mapping

We start by giving an idea of how the algorithm finds the Viper counterpart of a Rust variable. This works differently depending on the version of Prusti.

In Section 3.1.1 and Section 3.1.2 we assume we have a variable `x` with type `struct X { val: i32 }` and MIR name _1. Let us see how we find the corresponding snapshot variable.

3.1.1 Old version

As the snapshot-based type encoding is only used on demand in the old version, we might not be able to find a snapshot variable. If the encoding is used and a snapshot domain is constructed, there is also an associated heap-dependent function: `function snap_X(self: Ref): Snap_X { ... }`. This function links heap-dependent variables with snapshot variables.

In Listing 3.1 we have a heap-based variable and a snapshot variable. Both variables represent the same Rust variable. If we want to switch from the heap-based variable to the other one we simply call the heap-dependent function as seen in line 5.

```
method fail() {
    var _1: Ref
    var _1_snap: Snap_X
    ...
    _1_snap := snap_X(_1)
    ...
}
```

Listing 3.1: Heap-dependent function usage.

```
1 function snap_X(Ref): Snap_X {
2     _1 -> _1_snap
3     else -> #unspecified
4 }
```

Listing 3.2: Counterexample of heap-dependent function.

Assuming verification of `fail()` fails, Silicon will provide a counterexample for the heap-dependent function. Listing 3.2 shows the counterexample which contains the link between `_1` and `_1_snap` in line 2. Since we already know that `_1` corresponds to `x`, we have found a snapshot variable for `x`.

3.1.2 Refactored version

In the refactored version Rust variables are directly translated into snapshot variables. As explained in Section 2.3, Prusti uses SSA and we loose the one-to-one relationship between Rust and Viper variables. So instead of searching for a single snapshot variable we are searching for a set of variables. Let us look at an example:

```
method fail() {
    var _1_snap_0: Snap_X
    var _1_snap_1: Snap_X
    var _2_snap_0: Snap_X
    var _2_snap_1: Snap_X
    ...
}
```

Listing 3.3: Method with multiple snapshot variables.

What we see in Listing 3.3 is a method with various snapshot variables. We make use of the common prefix of these variable names to link them to MIR variables, e.g. `_1` $\rightarrow$ {`_1_snap_0`, `_1_snap_1`}. We utilize this set, a concrete counterexample and the control-flow graph of the MIR to extract a list of relevant snapshot variables. This part is quite challenging and will be explained in full detail in Chapter 4. At this point, we have found a list of snapshot variables for our `x`.

3.2 Back translation

Once we have found a snapshot variable for `x`, we can translate it back to the level of Rust. As already explained in Chapter 2, the snapshot-based encoding contains destructor functions. We use these functions combined with the type information of a Rust variable, e.g. which fields are part of it, to construct a counterexample. Let us assume we have failed to verify some method with Silicon and were presented with the following counterexample:

```
destructor_Snap_X_f_val(Snap_X) : Int {
    _1_snap_0 -> 1
    _2_snap_1 -> 2
    else -> #unspecified
}
```

Listing 3.4: Back translation.

Our goal is to find a counterexample for `x` by extracting a value for its field `val`. In Listing 3.4 we see the destructor function of our field. In this function we find two key-value pairs: (`_1_snap_0, 1`) and (`_2_snap_1, 2`). If we assume the snapshot variable of `x` is `_1_snap_0`, we learn that the value of our field `val` is `1`. If a value of a key-value pair is not a primitive type this process is continued recursively. The back translation is possible for Booleans, characters, integers, references, tuples, structs, enums, arrays and unions.

3.3 Reporting

At this point we have a final counterexample and want to report it back to the user. This is different depending on which version of Prusti is used. In addition to the standard format, we also provide a new customizable format for both versions.

We start with the old version. While the final output has not changed at all for the user, internally it works a bit differently. We have to understand that Prusti produces one counterexample for both type encodings. The user should receive the best possible combination of these two. This is achieved by a new merging process which iterates through the variables and creates the most informative counterexample possible.

In the refactored version the format completely changes. Instead of only providing an initial and a final value, we can now reflect all variable changes in the counterexample. The following listings demonstrates the difference between both versions:

```
struct X {
    val: i32,
}

#[ensures(x != 3)]
fn fail(x: X) -> i32 {
    x.val += 1;
    x.val += 2;
    x.val
}
```

Listing 3.5: Simple Rust function.

```
counterexample for "x"
    initial value:
        X { val: 1, }
    final value:
        X { val: 3, }
```

Listing 3.6: Part of the counterexample output in the old version. We see that a variable has an initial value and a final value.

```
counterexample for "x"
    value: X { val: 1, }
counterexample for "x"
    value: X { val: 2, }
counterexample for "x"
    value: X { val: 3, }
```

Listing 3.7: Part of the counterexample output in the refactored version. We see that each change is presented to the user individually.

Even though we want our counterexamples to be as detailed as possible, at some point, especially for nested types, they will be very hard to read and not helpful anymore. As a countermeasure we provide the user with an option to customize how types will be presented in the counterexample. This is done via a new procedural macro. The user can specify a custom format and choose which fields in which order should be included. As we will see later this improves the readability of more complex types massively.

```
#[print_counterexample("custom output of X: val = {}", val)]
struct X {
    val: i32,
}
```

Listing 3.8: Macro for custom counterexample in Prusti.

```
note: counterexample for "x"
    custom output of X: val = 1
```

Listing 3.9: Customized counterexample format.

Chapter 4

Implementation

In this chapter we take a detailed look into the implementation of the counterexample generation algorithm. Section 4.1 explains how information is extracted from functions in counterexamples from Silicon. What follows, in Section 4.2, is a detailed discussion of the back translation algorithm for both versions of Prusti. As a last point, in Section 4.3, we discuss the new procedural macro for customizing counterexamples.

4.1 Functions

In this section we explain how to extract information from functions. It is divided into three parts. The first part, Section 4.1.1, discusses the extraction process. In the second part, Section 4.1.2, we interpret the meaning of extracted information in terms of counterexamples and domains. The last and third part, Section 4.1.3 explains a special case, called the *default case*, in more detail.

4.1.1 Extraction process

In general, a function defines a relation between some input values and some output values. Whenever we are given an input, we can uniquely identify the function's output. Let us look at a general form of a function in a counterexample:

In order to extract information we store all entries from a function body in a map. In this map the key is defined by the list of types in the header and the value is defined by the return type. Once we have this map, extracting information from a function is equal to checking if a certain key exists in its map or not. If the key exists, we return the value of this map's entry. If the key does not exist, we return the default case.

```
function_name(type_1, ..., type_n) : type_re {

    (val_1 : type_1, ..., val_n : type_n)_1 -> val_re_1 : type_re
    ...
    (val_1 : type_1, ..., val_n : type_n)_k -> val_re_k : type_re

    else -> val_default : type_re  //default case
}
```

Listing 4.1: The function consists of a function header and a function body. The header is made up by a unique name, a list of types and exactly one return type. In the function body we always have exactly one default case and an arbitrary amount of entries. Each entry consists of a list of concrete values, where each value's type is defined by the list of types from the function header, and a single return value.

4.1.2 Interpretation

While general functions in a counterexample just capture a relation between inputs and outputs, domain functions can be interpreted in a different way. A domain function of the form

`function_name(a: Domain_name): return_type`

can be interpreted as a function that *destructs* a domain into a smaller subpart, similar to *getter* functions in programming languages like Java. Then, a counterexample of an instance of a domain type can be interpreted as all the extracted information from all these functions for a given snapshot variable.

```
domain Snap {
    function value(self: Snap) : Int
}
```

Listing 4.2: In this simple domain the function `value` defines the relation between an instance of a domain type and a smaller subpart of it.

In the domain `Snap` we find one function of the previously described form. Therefore a counterexample for variables of type `Snap`, can be interpreted as extracting the information from the function `value()`. In other words, a counterexample for variables of type `Snap` is isomorphic to a single integer.

4.1.3 Default value

An extraction process for the function `value` in Listing 4.2 either returns the value of an entry or it returns the default value. If it is the former we *definitely* have found a counterexample. If it is the latter we *might* have one.

If we use the extraction process on the function `value` for both variables `x` and `y`, we get `x = 0` and `y = 0`. This cannot be a valid counterexample for the method `fail` because it violates the precondition `y != 0`. This means we have found a counterexample for `x` but not for `y`. Therefore the default value only *might* return a counterexample.

```
method fail(x: Snap, y: Snap)
    returns (res:Int)
    requires value(y) != 0
    ensures res != 0
{
    res:= value(x)
}
```

Listing 4.3: A verification of this method will fail. The variable `y` is restricted by the precondition, but `x` is unrestricted and therefore the postcondition cannot be guaranteed.

```
value(Snap) : Int {
    else -> 0
}
```

Listing 4.4: This is part of the counterexample provided by Silicon. The function body only contains the default case and no other entries.

The problem is the generation of the default case. Whenever a counterexample for a function is generated, Silicon's SMT solver Z3 chooses arbitrarily one of the function's entries as the default case. While this is a highly simplified explanation, it explains why we gain a default case by losing one function entry.

This default case generation process can be changed by passing a flag called `model.partial = true` via Silicon to Z3. Now Z3 creates an unspecified default case and does not use an entry anymore.

```
value(Int) : Int {
    x -> 0
    else -> #unspecified
}
```

Listing 4.5: The new counterexample for Listing 4.3 created by Silicon with the flag `model.partial = true`. The function body contains an unspecified default case and an additional entry.

If we use our extraction process on the counterexample from Listing 4.5, we get `x = 0` and `y = unknown`, which is a valid counterexample for our method `fail()`.

4.2 Back translation

In this section we explain how Prusti translates a Silicon counterexample into its internal representation and how this representation gets reported back to the user. This process has to be explicitly enabled via the environment variable `PRUSTI_COUNTEREXAMPLE = true`.

Since the algorithm differs in the two versions, we split this section into two parts. Section 4.2.1 explains the translation for the old version and Section 4.2.2 explains the translation for the refactored version. It is worth mentioning here that in the following sections names of variables, functions, etc. have slightly been changed and shortened for better readability.

4.2.1 Old version

In the old version of Prusti the counterexample generation algorithm consists of three steps. Each step is repeated for each variable from the Rust program. The first step, called variable mapping, tries to find a snapshot variable for a Rust variable. The next step translates this snapshot variable into an internal representation of Prusti. This is explained in detail in the refactored version in Section 4.2.2 and skipped at this point. Instead we will talk about some limitations of the algorithm. The last section and final step is presenting the counterexample to the user.

Variable Mapping

Let us assume we want to find a snapshot variable for the Rust variable `x` of type `struct X { val: i32 }`. For simplicity we assume we already know that `_1` is the MIR variable name of `x`.

```
1  method fail() returns (_0: Ref)
2  {
3    var _1: Ref
4    ...
5    label start
6    ...           // Preconditions
7    label l0
8    ...
9    assert (... snap_X(_1))  //failing assertion
10   ...
11 }
```

Listing 4.6: The method `fail()` is a simplified Viper program generated by Prusti. It contains the variable _1 as a result of the heap-based type encoding. We see in line 9 how the snapshot function links the variable _1 to a snapshot domain. This is typically done in assertions.

```
snap_X(Ref): Snap_X {
    _1 -> snap_0
    else -> #unspecified
}
```

Listing 4.7: Counterexample for the snapshot function.

Given this information, there exists a Viper variable of type `Ref` called `_1`. As outlined in Chapter 3, we can use the snapshot function `snap_X(self: Ref) : Snap_X` to find a snapshot value.

If we extract the information from the snapshot function in Listing 4.7 we find the snapshot variable of `x`, in this case `snap_0`.

Limitations

We are going into detail about two limitations of the snapshot function that restrict the ability of the counterexample generation.

First, Prusti's heap-based counterexample algorithm produces an initial and a final value for each variable. The snapshot-based counterexample algorithm can only produce a single value because the snapshot function contains only one entry per variable. Since snapshot functions are called inside assertions, this entry corresponds to the final value of a variable. In the future this could be fixed by calling snapshot functions at multiple places, but for the moment we can only produce final values.

The second limitation that we are discussing is about the MIR. As explained in Chapter 2, the encoded Viper program is strongly influenced by the MIR of the Rust program. If, in the MIR, one variable is assigned to another variable, e.g. `_2 = _1`, this is also done in the Viper program.

```
1  field val_ref: Ref
2  method fail() returns (_0: Ref)
3  {
4    var _1: Ref
5    var _2: Ref
6    ...
7    label start
8    ... // Preconditions
9    label l0
10   ...
11   _1.val_ref := _2 //encodes an assignment in the MIR
12   ...
13   assert (... snap_X(_2))
14   ...
15 }
```

Listing 4.8: This method is similar to the one in Listing 4.6. In line 11 we see how an assignment is encoded in Viper. After this line the variable `_2` is used to represent `x` instead of `_1`. For that reason, the snapshot function is called with `_2` instead of `_1`.

Even though the change in Listing 4.8 is only minor, it has a major impact on the counterexample.

```
snap_X(Ref): Snap_X {
    _2 -> snap_0
    else -> #unspecified
}
```

The function's counterexample contains the entry `_2 -> snap_0` instead of the entry `_1 -> snap_0`. Prusti's counterexample algorithm does not know about the relation between `_1` and `_2`, and an attempt to find a snapshot variable for `_1` will fail and therefore a counterexample cannot be generated. Unfortunately, this problem cannot easily be fixed and for the time being we have to accept this limitation.

Presentation

Let us assume the previously mentioned limitations do not apply and we were able to translate a snapshot variable back to the level of Rust. At this point we have produced two counterexamples for our variable `x`, one with the heap-based type encoding and one with the snapshot-based type encoding. Before we can present a final counterexample to the user, the two counterexamples have to be merged.

The goal of this merging process is to collect as much information as possible. This is achieved by iterating through each entry of the heap-based counterexample and fill missing values with their counterparts in the snapshot-based counterexample.

```
heap-based counterexample:
    X {
        a: 1,
        b: ?,
    }
snapshot-based counterexample:
    X {
        a: ?,
        b: 2,
    }
```

Listing 4.9: We see a fictitious counterexample for a variable of type `struct X{ a: i32, b:i32 }`. The heap-based counterexample has a missing value for `b`. Fortunately, it can be replaced by the value produced by the snapshot-based counterexample.

```
merged counterexample:
    X {
        a: 1,
        b: 2,
    }
```

Listing 4.10: The merged counterexample for the type `struct X{ a: i32, b:i32 }` combines the information of each individual counterexample.

```rust
fn fail(mut x: i32, y: i32)
{
    x = 1;
    if (x == 1) {
        x = 2;
    } else {
        x = 3;
    }
    assert!(x == 3)
}
```

Listing 4.11: The function body contains a control sequence (if-else block) and three variable assignments for x. The final assertion will always fail regardless of concrete function arguments.

After we merged the counterexamples, as seen in Listing 4.10, the algorithm is finished. This concludes the extension of Prusti's counterexample generation for the old version.

4.2.2 Refactored Prusti

At the time of writing the refactored version has to be explicitly enabled via the environment variable `PRUSTI_UNSAFE_CORE_PROOF = true`. Similarly to the old version, the algorithm can be summarized in three steps, which are repeated for each variable in the Rust program. In the first step, we find a mapping to snapshot variables. After that we translate each snapshot variable back to Prusti's intermediate representation, and finally we report it back to the user.

Variable mapping

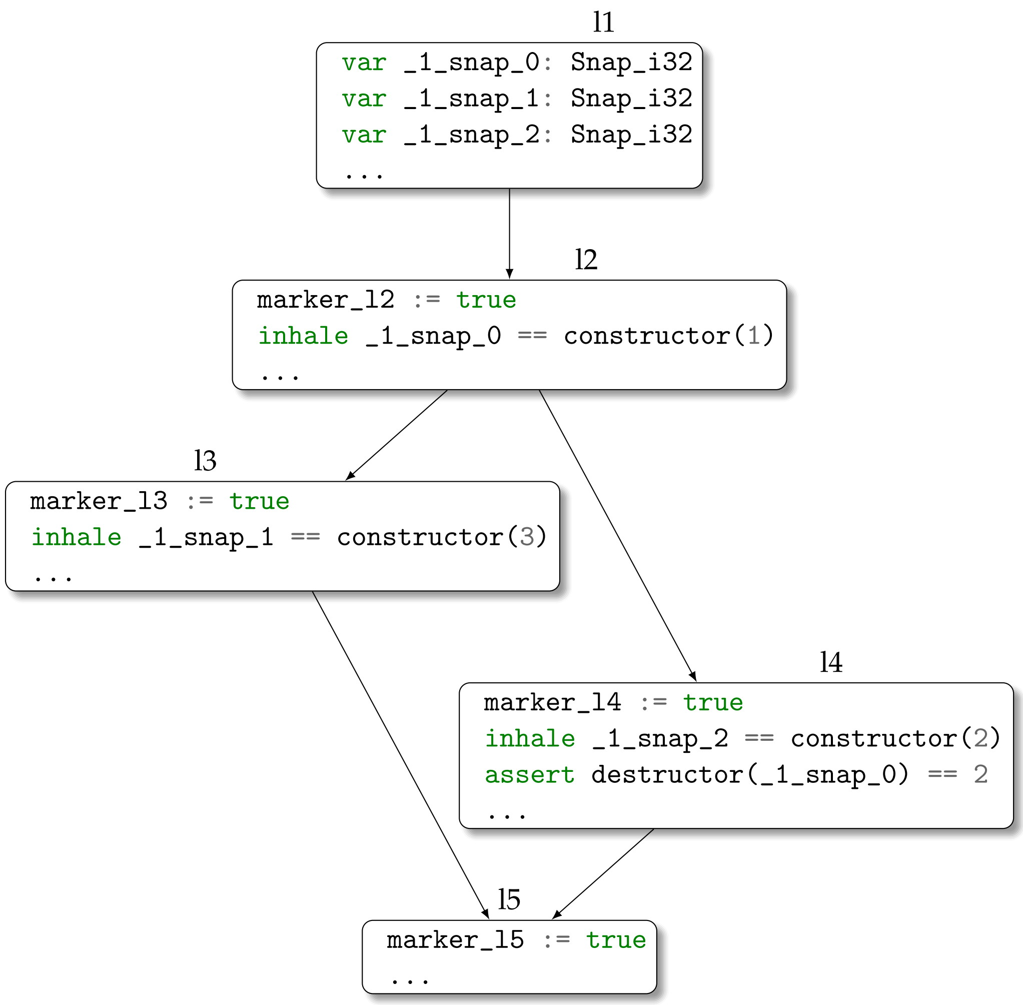
As outlined in Chapter 2 the refactored version directly encodes Rust variables into snapshot variables. In addition, it uses Static Single Assignment (SSA). This means whenever a Rust variable is changed, it will be encoded as a fresh snapshot variable. For that reason, we do not have a one-to-one relation between a snapshot and a Rust variable, and our objective changes to finding a list of snapshot variables instead of a single one.

We describe this process with an example. Let us assume we have a simple Rust function with two parameters, x and y, and want to find x's list of snapshot variables.

The Viper program in Listing 4.12 contains four snapshot variables. Not all of them are relevant for x. As a first step, we collect all variables with the prefix `_1` (the MIR variable name of x) and obtain {`_1_snap_0`, `_1_snap_1`, `_1_snap_2`}.

```
method fail()
{
  var _1_snap_0: Snap_i32
  var _1_snap_1: Snap_i32
  var _1_snap_2: Snap_i32
  var _2_snap_0: Snap_i32
  var marker_l2: Bool
  var marker_l3: Bool
  var marker_l4: Bool
  var marker_l5: Bool

  label l2
  marker_l2 := true
  //equal to x = 1
  inhale _1_snap_0 == constructor(1)
  //equal to if (x == 1)
  if (destructor(_1_snap_0) == 1) {
    goto l4
  }
  goto l3

  label l3
  marker_l3 := true
  //equal to x = 3
  inhale _1_snap_1 == constructor(3)
  goto l5

  label l4
  marker_l4 := true
  //equal to x = 2
  inhale _1_snap_2 == constructor(2)
  assert destructor(_1_snap_0) == 3
  goto l5

  label l5
  marker_l5 := true
}
```

Listing 4.12: This is a simplified version of the encoded Viper program for Listing 4.11. We used comments to indicate which line of Viper code corresponds to which part of the Rust source code.



```
l1
var _1_snap_0: Snap_i32
var _1_snap_1: Snap_i32
var _1_snap_2: Snap_i32
...
```

```
l2
marker_l2 := true
inhale _1_snap_0 == constructor(1)
...
```

```
l3
marker_l3 := true
inhale _1_snap_1 == constructor(3)
...
```

```
l4
marker_l4 := true
inhale _1_snap_2 == constructor(2)
assert destructor(_1_snap_0) == 2
...
```

```
l5
marker_l5 := true
...
```

Figure 4.1: Each basic block in the control-flow graph of Listing 4.12 is uniquely named. All blocks, except the first one, set their marker to `true` as their first statement. This indicates that if a basic block was visited or not in a concrete counterexample. The first basic block is always visited and therefore does not have to be marked.

Some elements from this set are relevant for a counterexample and some are not. As a next step, we have to filter and to order it. This is done via the control-flow graph (CFG) of the Rust program.

In each basic block of Figure 4.1 we search for *inhale* statements. In the context of counterexamples an inhale statement means that a Rust variable, in our case `x`, was assigned a new value. Since Prusti uses SSA, each snapshot variable occurs in exactly one inhale statement. We store this information in our set of snapshot variables as follows: {(`_1_snap_0, l2`), (`_1_snap_1, l3`), (`_1_snap_2, l4`)}.

Everything up to this point is done before the actual verification takes place.

```
marker_l2 <- true
marker_l3 <- false
marker_l4 <- true
marker_l5 <- false
```

Listing 4.13: Silicon produces this counterexample for Listing 4.12. Of course, the whole counterexample is much bigger, but at this point we are only interested in the markers. Each marker indicates whether its basic block was visited or not.

Since the next steps depend on a concrete counterexample, Prusti has to verify the program next. This provides us with a counterexample from Silicon.

We use the markers from Listing 4.13 to construct a path in the CFG. In this case the visited basic blocks are `l1`, `l2` and `l4` in that particular order. This knowledge is enough to filter and order our previously obtained set, and we get the desired list of snapshot variables: `(_1_snap_0, _1_snap_2)`.

Note that the list captures all changes of `x`. In general, this is much more detailed than the mapping in the old version. For that reason, we are able to produce more fine-grained counterexamples.

Types

Once we have found a list of snapshot variables, we have to translate them back into a Prusti-internal representation. The representation is an enum. Each variant represents a support type for counterexamples.

While in the old version we only translated a single snapshot variable back to the level of Rust, in the refactored version the variable mapping produced a list of snapshot variables. Therefore the translation algorithm will also produce a list of translated variables.

For the following explanation we assume we have the list `(_1_snap_0, _1_snap_1)` and want to translate it into Prusti's intermediate representation.

Primitive types We consider integers, Booleans and characters as primitive types. Integers and Booleans have direct counterparts in Viper, while characters have to be converted to integers first. Apart from that, they all work analogously.

For a Rust type like `i32`, the snapshot-based type encoding produces a domain called `Snap_I32`. This domain contains a function called `destructor_Snap_I32_value(value: Snap_I32): Int`. This function "destructs" the domain type and gives us our desired value of type `Int`.

In Listing 4.14 we see a fictitious counterexample of the destructor function. Extracting information from this function for each element of `(_1_snap_0,`

```
destructor_Snap_I32_value(Snap_I32): Int {
    _1_snap_0 -> 1
    _1_snap_1 -> 2
    _2_snap_1 -> 3
    else #unspecified
}
```

Listing 4.14: Silicon counterexample for the Rust type `i32`.

```
domain Snap_I32 { ... }
domain Snap_ref_I32 {
  function destructor_Snap_ref_I32_target_final
    (self: Snap_ref_I32): Snap_I32

  function destructor_Snap_ref_I32_target_current
    (self: Snap_ref_I32): Snap_I32
  ...
}
```

Listing 4.15: These are the domains created by the snapshot-based type encoding for the type `&i32`.

`_1_snap_1`) results in the following list of Prusti's internal representation: `[Int('1'), Int('2'))]`.

References If the type to be translated is a reference in Rust, e.g. `&i32`, the snapshot-based type encoding produces two domains, one for the underlying type, in this case `i32`, and one for the actual reference.

The domain `Snap_ref_I32` in Listing 4.15 contains two different destructor functions. Both functions return a value of type `Snap_I32`. Therefore we have to apply the destructor twice. First, we use one of the two functions from Listing 4.15 to obtain a new snapshot variable with type `Snap_I32`. Second, we destruct the domain `Snap_I32` as previously described.

To understand which function we should use, we have to look at a concrete counterexample.

In Listing 4.16 we see that the function *target_final* returns the same snapshot variable for both `_1_snap_0` and `_1_snap_1`, while the other function returns different snapshot variables. In other words, the former function returns a final snapshot variable and the latter returns intermediate snapshot variables. Since we want intermediate variables, we have to use the function *target_current*.

Assuming the destructor function of `Snap_I32` contains the entries `_3_snap_2` → `1` and `_3_snap_1` → `2`, the translated counterexample looks like this: `[Ref(Int('1')), Ref(Int('2')))]`.

```
destructor_Snap_ref_I32_target_final(Snap_ref_I32): Snap_I32 {
    _1_snap_0 -> _3_snap_2
    _1_snap_1 -> _3_snap_2
    _2_snap_1 -> _3_snap_1
    else #unspecified
}
destructor_Snap_ref_I32_target_current(Snap_ref_I32): Snap_I32 {
    _1_snap_0 -> _3_snap_1
    _1_snap_1 -> _3_snap_2
    _2_snap_1 -> _3_snap_1
    else #unspecified
}
```

Listing 4.16: Silicon provides counterexamples for both destructor functions for references.

```
domain Snap_I32 { ... }
domain Snap_struct_X {
  function destructor_Snap_struct_X_val
    (self: Snap_struct_X): Snap_I32
  ...
}
```

Listing 4.17: These are the domains created by the snapshot-based type encoding for the type `struct X { val: i32 }`.

Tuples and structs Tuples and structs work in a very similar manner, we can look at them together. A tuple can be thought of as a struct with unnamed fields.

Let us assume we want to translate the Rust type `struct X { val: i32 }`. The snapshot-based type encoding generates a domain for the struct itself and domains for all types occurring in the fields.

In Listing 4.17 we see the destructor function for our field `val`. In general, there is a destructor function for each field. While for named fields those function names end with the field name, in the case of unnamed fields those function names end with indices. Apart from that they are equivalent and are used in the exact same way.

We translate a struct by translating each field. The destructor function of a field returns a new snapshot variable which has to be translated as well. Therefore the whole process is recursive.

```
[ Struct {
    name: "X",
    field_entries: [val: Int('1')],
 },
 Struct {
    name: "X",
    field_entries: [val: Int('2')],
 }]
```

Listing 4.19: Prusti's counterexample entry for a struct.

```
destructor_Snap_struct_X_f_val(Snap_struct_X): Snap_I32 {
    _1_snap_0 -> _3_snap_1
    _1_snap_1 -> _3_snap_2
    else #unspecified
}
```

Listing 4.18: Silicon counterexample for struct.

In Listing 4.18 we see a counterexample for our struct. The translation of the field `val` gives us two new snapshot variables. If we assume those can be recursively translated, we end up with the following counterexample:

Enums and Unions We can view an enum as a collection of structs. Each element of this collection is called a variant. At run time exactly one variant is active. If we want to translate an enum, we have to figure out which variant is currently active.

Rust does this internally by storing a *discriminant*. A discriminant is an integer, which indicates which variant is active. This is also encoded in Viper, and we can use a special function to determine the discriminant.

Even though unions and enums are different in Rust, the snapshot-based type encoding encodes both types in the same way. Therefore their translation process is identical.

Let us assume we have to translate the type `enum X { First, Second(i32) }`. The snapshot-based type encoding will create a domain for the enum itself and a domain for each variant. Since variants are struct-like, it will also create more domains for all types occurring in fields.

We already explained how structs are translated, and therefore we are only interested in finding the correct variant. For that purpose, in Listing 4.20 we only look at the domain for the enum itself and omit the rest.

In Listing 4.20 we have three relevant functions in the domain `Snap_enum_X`. The first two functions destruct our enum to a specific variant. The third

```
domain Snap_enum_X {
  function destructor_Snap_enum_X_first
    (self: Snap_enum_X): Snap_First

  function destructor_Snap_enum_X_Second
    (self: Snap_enum_X): Snap_Second

  function discriminant_Snap_enum_X
    (self: Snap_enum_X): Int
  ...
}
```

Listing 4.20: This is the domain created by the snapshot-based type encoding for the type `enum X { First, Second(i32) }`.

```
discriminant_Snap_enum_X(Snap_enum_X): Int {
    _1_snap_0 -> 1
    _1_snap_1 -> 2
    else #unspecified
}
destructor_Snap_enum_X_first(Snap_enum_X): Snap_First {
    _1_snap_0 -> _4_snap_1
    else #unspecified
}
destructor_Snap_enum_X_Second(Snap_enum_X): Snap_Second {
    _1_snap_2 -> _3_snap_0
    else #unspecified
}
```

Listing 4.21: Silicon counterexample for an enum.

function supplies us with the discriminant.

The translation process works in the following way: First, we extract the discriminant. Depending on the discriminant, we choose a destructor function and destruct the domain `Snap_enum_X`. This gives us a new snapshot variable, which can be recursively destructed as described earlier.

Since the snapshot variable `_1_snap_0` in Listing 4.21 has a discriminant of `1`, we destruct it to the domain `Snap_struct_First`. The snapshot variable `_1_snap_1` has a discriminant of 2 and we destruct it to the domain `Snap_struct_Second`. Both of those domains represent structs which can be translated recursively. Assuming this translation was successful, a counterexample could look like this:

```
[ Enum {
    super_name: "X",
    name: "First",
    fields_entries: [],
  },
  Enum {
    super_name: "X",
    name: "Second",
    fields_entries: [0: Lit('2')],
  }]
```

Listing 4.22: Prusti's counterexample entry for an enum.

```
domain Snap_I32 { ... }
domain Snap_struct_box_I32 {
  function destructor_Snap_struct_box_I32_val_ref
    (self: Snap_struct_box_I32): Snap_I32
  ...
}
```

Listing 4.23: These are the domains created by the snapshot-based type encoding for the type `Box<i32>`.

```
destructor_Snap_struct_box_I32_val_ref(Snap_struct_box_I32): Snap_I32 {
    _1_snap_0 -> _3_snap_1
    _1_snap_1 -> _3_snap_2
    else #unspecified
}
```

Listing 4.24: Silicon counterexample for box.

Box Even though technically this type is a struct, it is handled differently in Rust and as a consequence in Prusti. It has to be translated differently. Note that at the time of writing this type is not fully supported in the refactored version. Nonetheless, we still want to provide a counterexample.

We begin by looking at the domains generated by the snapshot-based type encoding for the Rust type `Box<i32>`.

As we can see in Listing 4.23 the Viper domain of a box type contains a different destructor function than a domain of a normally encoded struct type would have. Nonetheless, we can use this function to destruct the domain to get a new snapshot variable. The domain type of this new snapshot refers to the inner type of the box.

The destructor function in Listing 4.24 returns snapshot variables of the type `Snap_I32`. Assuming those can be translated, the counterexample looks like this: `[Box(Int('1')), Box(Int('2'))]`.

```
_1_snap_0 <- [_3_snap_1, _3_snap_1]
_1_snap_1 <- [_3_snap_0, _3_snap_0, _3_snap_0]
```

Listing 4.25: Silicon counterexample for a sequence.

```
domain Functions {
  function m_foo(_1: Snap_I32, _2: Snap_I32, ... ): Snap_I32
  ...
}
function caller_for_m_foo(_1: Snap_I32, _2: Snap_I32): Snap_I32
    requires ...
    ensures ...
{
    m_foo(_1, _2, ...)
}
```

Listing 4.26: A pure function in Rust has a counterpart in the `Functions` domain in Viper. This domain function has an additional parameter which is not relevant for counterexamples and will be omitted. Whenever a pure function is called in Rust, the function `caller_for_m_foo()` is called in Viper.

Sequences and arrays A sequence is a mathematical type available in Prusti's specifications and has a direct counterpart in Viper. An array is encoded in a similar way as a sequence and therefore translated in an identical manner.

The domain for a sequence or array created by the snapshot-based type encoding does not contain any destructor functions. A counterexample has to be extracted differently.

The type of such a Viper variable will not be of a domain. Instead it will be a Viper sequence of a domain. For example, if some variable `x` is of Rust type `Seq<i32>`, its Viper variables are of type `Seq[Snap_I32]`. A Silicon counterexample for these Viper variables will, of course, look different.

In Listing 4.25 we see that a counterexample of a sequence does not contain any functions. Instead a snapshot variable, e.g `_1_snap_0`, directly contains a list of new snapshot variables. Each of these new snapshot variables must be translated individually. Assuming `_3_snap_0` and `_3_snap_1` are translated to `1` and `2`, respectively, the translated counterexample looks like this: `[Seq([Int('2'), Int('2')]), Seq([Int('1'), Int('1'), Int('1')])]`.

Pure functions In addition to variables we also include pure functions in our counterexamples. A pure function is encoded as two Viper functions, a heap-dependent one and a heap-independent one. Given a pure Rust function `foo(x: i32, y: i32) -> i32`, its encoding is shown in Listing 4.26.

In order to generate a counterexample for pure functions we need to find

```
caller_for_m_fn_name(heap, Snap_I32, Snap_I32): Snap_I32 {
    heap_0 1_snap_0 1_snap_2 -> 1_snap_3
    heap_1 2_snap_1 1_snap_1 -> 1_snap_4
    else #unspecified
}
```

Listing 4.27: Silicon counterexample for a pure function. The `caller_for_m_fn_name()` is a heap-dependent function and has an additional parameter referring to a heap location. Since our snapshot variables are heap-independent, the return value of the function does not change if the heap location changes. Therefore the heap variable has no impact on the counterexample and we can ignore it. We treat the function as if it only has two parameters.

all relevant `caller_for_m_foo()` calls. This is equivalent to finding a variable mapping, but instead of searching for snapshot variables we search for `caller_for_m_foo()` functions and store all the arguments. Let us call this process function mapping.

If we assume that the function `foo()` is called twice in the original Rust program, the function mapping could find the following list for `foo()`: `([1_snap_0, 1_snap_2], [2_snap_1, 1_snap_1])`.

We see that the function mapping is a list that contains lists of snapshot variables. Each inner list is a collection of all arguments used in a particular function call.

Given a counterexample for the function `caller_for_m_foo()` like in Listing 4.27, we can translate a function in the same manner as variables Assuming `1_snap_3` and `1_snap_4` are translated to `1` and `2`, respectively, the translated counterexample looks like this: `[Int('1'), Int('2')]`.

4.2.3 Presentation

Up to this point we have only produced a list with Prusti's internal representation, one for each Rust variable. The final step is presenting this to the user.

We decided to format the counterexample similarly to the old version. The only difference is that each variable has an arbitrary number of values instead of just an initial and a final value.

We are demonstrating this with the following example: The function in Listing 4.28 has an immutable variable `x` and a mutable variable `y`. It is clear that a single value in a counterexample for the variable `x` is sufficient since it cannot change its value. The variable `y` is assigned twice and therefore two values are can be shown in a counterexample.

The major advantage and the reason for its similarity to the old version is its compatibility with the VSCode extension Prusti-Assistant [5]. The coun-

```rust
#[requires(x > 0)]
#[ensures(result != 2)]
fn fail (x: i32) -> i32 {
    let mut y = x;
    y = y + 1;
    y
}
```

Listing 4.28: Rust function with multiple assignments.

```
note: counterexample for "x"
       value:   1
  --> output.rs:9:10
   |
9  | fn fail (x: i32) -> i32
   |          ^
note: counterexample for "y"
       value:   1
  --> output.rs:10:17
   |
10 |     let mut y = x;
   |                 ^
note: counterexample for "y"
       value:   2
  --> output.rs:11:5
   |
11 |     y = y + 1;
   |     ^^^^^^^^^
note: counterexample for "result"
       value:   2
  --> output.rs:12:5
   |
12 |     y
   |     ^
```

Listing 4.29: A counterexample for Listing 4.28. While the variable x only has one value, the variable y has two. In addition, each note is annotated by a line number to refer to the Rust code.

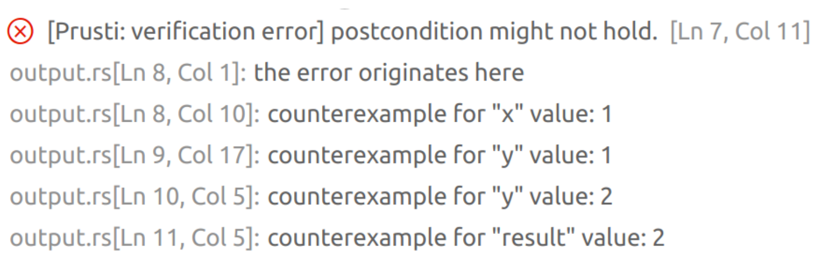

```
[Prusti: verification error] postcondition might not hold.  [Ln 7, Col 11]
output.rs[Ln 8, Col 1]: the error originates here
output.rs[Ln 8, Col 10]: counterexample for "x" value: 1
output.rs[Ln 9, Col 17]: counterexample for "y" value: 1
output.rs[Ln 10, Col 5]: counterexample for "y" value: 2
output.rs[Ln 11, Col 5]: counterexample for "result" value: 2
```

Figure 4.2: Counterexample output in Prusti assistant.

```rust
struct X {
    val: Vec<i32>,
}

#[model]
struct X {
    val: Seq<i32>,
}
```

Listing 4.30: A simple model which replaces the complicated type `Vec<i32>` with a much simpler mathematical type `Seq<i32>` in the form of a model.

terexample can be compactly presented as seen in Figure 4.2. The user can simply click on a note and jump to the correct code line.

4.2.4 Type models

In this section we explain how we provided counterexample support for type models. For that purpose, we have to understand how such models are encoded in Viper. Let us consider a simple model in Rust:

We keep a list of all models in Prusti. This allows us to check easily whether a given type has a model or not. Whenever this is the case we want to translate the model type instead of the original type. This requires translating a different snapshot variable.

In order to understand how we find this other snapshot variable, we have to take a look at the snapshot-based type encoding of a model.

As we can see in Listing 4.31 the snapshot-based type encoding will create four relevant domains:

- `Snap_struct_X`: This is the encoding of the original Rust type.
- `Snap_model_X`: This is the encoding of the model type.
- `Snap_ref_X`: This is the encoding of a reference to the original Rust type, in this case `&X`.

```
domain Snap_struct_X { ... }
domain Snap_model_X { ... }
domain Snap_ref_X {
  function constructor_Snap_ref_X
    (self: Snap_struct_X): Snap_ref_X
    ...
}
domain Functions {
    function to_model_X
        (model_ref: Snap_ref_X) : Snap_model_X
}
```

Listing 4.31: These are all the domains created by the snapshot-based type encoding for the example in Listing 4.30.

- `Functions`: This domain contains various heap-independent functions. The function relevant for us is called `to_model_X`.

Each snapshot variable of type `Snap_struct_X` has a counterpart of type `Snap_model_X`. Let us assume the former is called `_1_snap_0` and the latter is called `_2_snap_0`. In Viper this relation is encoded in the following way:

```
_2_snap_0 := caller_to_model_X(constructor_Snap_ref_X(_1_snap_0))
```

If we want to extract a counterexample for a model while only knowing the snapshot variable of the original type, we have to traverse these function calls to find the desired snapshot variable.

```
constructor_Snap_ref_X(Snap_struct_X): Snap_ref_X {
    _1_snap_0 -> _2_snap_0
    else #unspecified
}
caller_to_model_X(Snap_ref_X): Snap_model_X {
    _2_snap_0 -> _3_snap_0
    else #unspecified
}
```

Listing 4.32: Silicon's counterexample for the model from Listing 4.30.

Given the counterexample from Listing 4.32, we see how we find the desired snapshot variable `_2_snap_0` by only knowing `_1_snap_0`.

The last remaining step is to translate this new snapshot variable with the Rust type definition of the model back to a Prusti-internal representation. Unfortunately, this would always result in an unknown counterexample. This is due to the fact that destructor functions of domains which encode models are never explicitly called by Prusti encoded Viper programs. For that reason Silicon cannot provide a counterexample and as a result we cannot either.

This problem can be solved by explicitly forcing Prusti to call destructor functions. For that purpose we provide a new flag called `PRUSTI_UNROLL_MODEL = x`, where `x` is an integer which denotes the *depth* of a type. We define the depth of a type as how many destructor functions have to be called to fully translate a type, e.g. the type `i32` (domain `Snap_I32`) has depth 1 while the type `&i32` (domain `Snap_ref_I32`) has depth 2.

While this adds more complexity to the Viper program, which is not optimal, it is justifiable for small `x`. Since the purpose of models is mainly to simplify complex types, it is a reasonable assumption that the depth of its type is relatively small.

Note that the described process replaces a counterexample of the original type by a counterexample of its model. While this is the desired behaviour in most cases, it might be the case that the user needs both counterexamples. For cases like this, we provide a new flag called `PRUSTI_PRINT_COUNTEREXAMPLE_IF_MODEL_IS_PRESENT = true`. This flag forces Prusti to produce two counterexamples, one for the original type and one for the model type.

4.3 Customizable Counterexample

In this section we describe a new customizable format of a counterexample. We provide a new procedural macro attribute called `print_counterexample()`, which can be used to annotate structs and enums.

Let us revisit a recursive type definition for a `LinkedList`. Assuming we have some fictitious counterexample with three elements. The standard format of Prusti's counterexample output is very verbose, but it can be hard-to-read for e.g. a `LinkedList`.

The counterexample in Listing 4.33 already measures 15 lines of text with just three elements. Once the number of elements increases it will get even more difficult to read.

Our procedural macro `print_counterexample()` allows the user to omit irrelevant information and format the relevant information in a different way. Its syntax is a simplified version of the *formatted print* in Rust and should not be too alien for the user. Listing 4.35 and Listing 4.36 show an example of how a counterexample for the same LinkedList could look like.

```
counterexample for "list"
  value:   LinkedList {
    val: 0,
    next: Some(
      box( LinkedList {
        val: 1,
        next: Some(
          box( LinkedList {
            val: 2,
            next: None,
          }),
        ),
      }),
    ),
  }
```

Listing 4.33: Prusti's counterexample for a `LinkedList` with three elements.

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

Listing 4.34: Simple recursive type definition for a `LinkedList`.

```rust
#[print_counterexample("[{}] -> {}", val, next)]
struct LinkedList {
    val: i32,
    next: Option<Box<LinkedList>>,
}
```

Listing 4.35: Custom counterexample macro in Prusti.

```
counterexample for "list"
    value: [0] -> Some( box([1] -> Some( box([2] -> None ))))
```

Listing 4.36: Customized counterexample for a `LinkedList`.

4.3.1 Syntax

In this section we describe the syntax for the procedural macro. It can only be used with structs and enums. Since the syntax is slightly different for the two types, we explain it separately.

Struct If a struct is annotated, the macro must have at least one argument and the first argument must be of type `String` and can contain an arbitrary number of curly brackets. The number of curly brackets must match the number of the remaining arguments. The remaining arguments must either be a field name, if the fields are named, or an index, if the fields are

```
#[print_counterexample()]
 enum Name {
     #[print_counterexample(...)]
     Variant_1(...),
     Variant_2(...)
     ...
 }
```

Listing 4.38: The general form of the procedural macro for enums. The outer annotation must not contain any arguments. Each variant can be annotated as if it were a struct.

unnamed. A field can be used multiple times.

```
#[print_counterexample("... {} ...", var_i, ...)]
 struct Name {
     var_1: typ_1,
     ...
     var_n: typ_n,
 }
```

Listing 4.37: The general form of the procedural macro for structs.

Enums If an enum is annotated, the macro must not contain any arguments. Each variant can be annotated in the exact same way as previously described. Only annotating a variant without the enum itself will result in a compile time error.

4.3.2 Implementation

In this section we explain the implementation of this procedural macro.

Rust procedural macros are executed in a separate sandboxed process. This makes it difficult to pass data to the rest of Prusti's verification pipeline. Information has to be stored in specially created *dummy* functions and then revisited at a later stage. The reason why we use such functions is to make the compiler type check expressions for us.

The design process of our dummy function was driven by three main factors. First, it had to be able to encode all necessary information, e.g. arguments of the macro. Second, it should be able to detect syntax errors as early as possible. It would be very wasteful to verify a Rust program and then fail to produce a counterexample because of a syntax error. Third, the function should work in a similar fashion for both structs and enums.

In Listing 4.40 we can see how the procedural macro is encoded. Note that if the type would be a struct instead of a variant of an enum, only the if

```
#[print_counterexample()]
 enum X {
     #[print_counterexample("a = {}, b = {}", a, b)]
     V1 { a :i32, b: i32 },
     #[print_counterexample("V2.0 = {}, V2.1 = {}", 0, 1)]
     V2(i32, i32, i32),
     V3(i32),
 }
```

Listing 4.39: In this example we have two variants annotated with `print_counterexample()`. Prusti will internally produce two dummy functions to encode the macro for the first two variants and ignore the third one.

```
impl X{
    fn print_X_V1(self) {
        if let X::V1{a, b, ..} = self {
            "a = {}, b = {}"; a; b;
        }
    }
    fn print_X_V2(self) {
        if let X::V2{..} = self {
            "V2.0 = {}, V2.1 = {}"; 0; 1;
        }
    }
}
```

Listing 4.40: The example in Listing 4.39 will be encoded using these two dummy functions. Each function contains an if block to explicitly cast the type to a specific variant. This allows the compiler to check for type errors.

condition slightly changes. The major advantages of this form is that the compiler can automatically check for type errors.

The arguments of the macro are written inside the if block. Once a counterexample of type `enum X { ... }` should be produced, the arguments can be extracted and the format of the counterexample can be changed accordingly.

Chapter 5

Evaluation

In this chapter we evaluate our work. In Section 5.1 we compare our two versions of Prusti's counterexample generation algorithm with the original version. After that, in Section 5.2, we test the capabilities and limitations of the refactored version against complex data structures. Finally, in Section 5.3, evaluate the performance of the refactored version.

5.1 Comparison

In this section we compare our two extensions to the counterexample generation algorithm with the original version. In the thesis of the original counterexample algorithm [10], a collection of handpicked functions where chosen to evaluate its weaknesses and strengths. We are using the same collection of functions to showcase our improvements.

In the following, we will not discuss the details of these functions since this is already been done in the original thesis. Instead we talk about the changes in the counterexample.

Table 5.1 summarizes the findings of our comparison. We can see that the refactored version is able to improve the counterexamples in all cases except the last one, while the improved old version only provided a better counterexample in the case `account-fail.rs`.

We begin by discussing the counterexample of `tuple.rs`. Listing 5.1 and Listing 5.2 show the counterexample of the improved old version and of the refactored version respectively. We can see that the improved old version cannot produce an initial value for the variable `x`. This is due to the heap-dependence of the variable x and its missing permissions at verification failure. This is discussed in detail in Section 2.2.3. Since no snapshot-based

Example	Original ver.	Improved old ver.	Refactored ver.	LOC
sum.rs	complete	complete	complete	10
replace.rs	complete	complete	complete	13
tuple.rs	partial	partial	complete	8
account.rs	partial	partial	complete	18
enum.rs	partial	partial	complete	13
account-fail.rs	non	partial	complete	11
loop.rs	spurious	spurious	spurious	11

Table 5.1: The table compares the counterexample of all functions used in the original evaluation for the original version, the improved old version and the refactored version. A generated counterexample can either be *complete*, i.e. all variables are assigned some values, or it can be *partial*, i.e. some variables are assigned some values, or it can be *non*, i.e. no variables are assigned any value, or it can be *spurious*, i.e. assigned values are incorrect. The table also shows the lines of code (LOC).

type encoding is used, the improved old version and the original version generate the same counterexample.

The refactored version is able to provide a complete counterexample. Since the variable x is immutable, it is sufficient to only generate a single entry for x. The rest of the counterexample is similar to the other versions.

```
counterexample for "x"
    initial value: ( 1, ? )
    final value:   ( 1, 'c' )
counterexample for "y"
    final value:   -1
counterexample for "z"
    final value:   'c'
counterexample for "result"
    final value:   ( 'c', -1 )
```

Listing 5.1: Counterexample provided by the improved old version of Prusti for `tuple.rs`.

```
counterexample for "x"
    value: ( 1, 'c' )
counterexample for "y"
    value:   -1
counterexample for "z"
    value:   'c'
counterexample for "result"
    value:   ( 'c', -1 )
```

Listing 5.2: Counterexample provided by the refactored version of Prusti for `tuple.rs`.

The improved old version of Prusti does only provide partial counterexamples for `account.rs` and `enum.rs` for similar reasons. We will not go into more detail and discuss the program `account-fail.rs` instead.

To understand the counterexample we have to look at the program code in Listing 5.3. The function under verification is called `has_money()`. In its body it calls a *pure* function.

```rust
pub struct Account {
    balance: i32,
}
#[pure]
fn get_balance(acc: Account) -> i32 {
    acc.balance
}
#[ensures(result)]
fn has_money(acc: Account) -> bool {
    get_balance(acc) > 0
}
```

Listing 5.3: account-fail.rs.

There are two reasons why the original version cannot produce a counterexample. First, the original version does not support pure functions for counterexamples and second, the variable `acc` is not accessible anymore at the point of verification failure.

The improved old version of Prusti can generate a partial counterexample because the snapshot-based type encoding is used. As we have already discussed, a counterexample generated via the snapshot-based type encoding can only produce a final value. The counterexample can be seen in Listing 5.4.

The refactored version does support counterexamples for pure functions. Therefore, in addition to generating a counterexample for the variable `acc`, it also generates a counterexample for the function `get_balance()`. The counterexample can be seen in Listing 5.5.

```
counterexample for "acc"
    initial value:
        Account { balance: ? }
    final value:
        Account { balance: 0 }
```

Listing 5.4: Counterexample provided by the improved old version of Prusti for `account-fail.rs`.

```
counterexample for "x"
    value:
        Account { balance: 0 }
counterexample for
  "get_balance()"
    value:   0
```

Listing 5.5: Counterexample provided by the refactored version of Prusti for `account-fail.rs`.

The last program we talk about is `loop.rs`. As the name suggests, it contains a loop. This loop is annotated with a weak loop invariant. This weak loop invariant leads to an over-approximation and a verification failure even though the functions should succeed verification. In cases like this a counterexample is spurious. Since the counterexample depends on the wrong

```
#[trusted]
struct VecWrapper<i8> {
    values: Vec<i8>,
}

#[model]
struct VecWrapper<i8> {
    values: Seq<i8>,
}

impl VecWrapper<i8> {
    #[trusted]
    #[requires(self.model().values.len() > Int::new(index))]
    #[ensures(self.model().values[Int::new(index)] == result)]
    fn lookup(&self, index: i64) -> i8 {
       self.values[index as usize]
    }
}
```

Listing 5.6: `VecWrapper` is a wrapper for the Rust type `Vec`. Since the wrapper is `trusted`, its field cannot be accessed directly. For that reason the `trusted` function `lookup()` is implemented. This function returns the element corresponding to the given index. The model is used inside the specifications to give meaning to this function.

verification result, all three versions of Prusti produce an equally wrong counterexample. Unfortunately, this is a general problem of program verification and out of scope for this work.

5.2 Complex data structures

In this section we evaluate the capabilities and the limitations of the counterexample generation for the refactored version in terms of complex data structures. For that reason we have picked out two data structures and explain the resulting counterexamples for some functions.

5.2.1 Vector

The first data structure we are investigating is the Rust type `Vec`. Even though this type is not supported by Prusti, we can bypass this problem by defining a *trusted wrapper* and implement *trusted functions* to access its field. A model is used to abstract the wrapper such that it can be used in the function's specifications.

In Listing 5.6 we see an example on how this can be accomplished. It is worth noting that the `VecWrapper` can be defined with any arbitrary copy-

```
#[requires(v.model().values.len() == Int::new(4))]
fn sum(v: &VecWrapper<i8>) {
    assert!(v.lookup(0) + v.lookup(1) + v.lookup(2)
        + v.lookup(3) == 15)
}
```

Listing 5.7: Simple function using the `VecWrapper`.

```
assertion might fail with "attempt to add with overflow"
counterexample for "v"
    value: ref(VecWrapper_model {
                values: Seq(94, 34, ?, ?, ),
           })
```

Listing 5.8: Counterexample provided by the refactored version of Prusti for function `sum()`.

able type, but for simplicity we chose `i8`. For our example we only need the function `lookup()`, but the implementation can be easily extended with other functions if needed.

Next, let us consider a function which uses our previously defined `VecWrapper` and examine the counterexample. In Listing 5.7 we see a simple function that checks whether the sum of the vector's elements is equal to 15.

Of course the verification will fail and a counterexample is seen in Listing 5.8. Unfortunately, Prusti only provides us with a partial counterexample. Nevertheless, this counterexample is informative enough to reconstruct the error of verification. We can see that adding `94` and `34` is larger than `127` and therefore will overflow `i8`.

In general, Prusti is only able to provide partial counterexamples for sequences. This is due to the fact that Silicon only provides counterexamples for Viper sequences at accessed indices. In most cases this is not a problem because it is still enough to reconstruct the error.

Another example to demonstrate this would be preventing overflow in function `sum()`. This could be done by adding a precondition that forces each element of the VecWrapper's field `values` to be between zero and ten. In this case a counterexample would now depend on all elements and the fact that they do not add up to `15`. As we can see in Listing 5.9 Prusti does now provide a complete counterexample.

5.2.2 Binary tree

The second data structure we are investigating is a binary tree:

```
struct BinaryTree<i8> {
    val: i8,
the asserted expression might not hold
counterexample for "v"
    value: ref(VecWrapper_model {
                values: Seq(7, 5, 2, 3, ),
           })
```

Listing 5.9: Counterexample provided by the refactored version of Prusti for function `sum()` with added precondition.

```rust
struct BoxWrapper<T> {
    value: Box<T>,
}

impl<T> BoxWrapper<T> {
    #[trusted]
    fn new(value: T) -> Self {
        Self { value: Box::new(value) }
    }
    #[trusted]
    fn deref(&self) -> &T {
        &self.value
    }
}
```

Listing 5.10: The `BoxWrapper` is similar to what we have seen in the previous example about vectors. The major difference is that functions do not have pre-/postconditions.

```rust
    left: Option<Box<BinaryTree<i8>>>,
    right: Option<Box<BinaryTree<i8>>>,
}
```

This is a recursive data structure and therefore needs the Rust type `Box`. Unfortunately, at the time of writing the *dereferencing* operation was not implemented. We can again bypass this problem by defining a wrapper and implement functions as seen in Listing 5.10.

We have to leave these functions underspecified because of the missing implementation. This means that Prusti might fail to verify a correct program. An example of this is in Listing 5.11. Whenever one of the underspecified functions, `deref()` or `new()`, is called Prusti has no information about the return value and assumes an arbitrary one. This is reflected in the counterexample as well.

Therefore Prusti is not able to produce meaningful counterexamples for recursive types for the time being. However, certain properties of recursive types can also be described via *pure* functions.

Verification of the function `unbalanced` from Listing 5.12 results in the coun-

```
#[ensures(result == val)]
fn correct(val: i8) -> i8{
    let b = BoxWrapper::new(val);
    *b.deref()
}
```

```
postcondition might not hold.
counterexample for "val"
    value:   0
counterexample for "b"
    value:   BoxWrapper {
                value: box(1),
             }
counterexample for "v"
    value: ref(VecWrapper_model {
                values: Seq(7, 5, 2, 3, ),
           })
counterexample for "result"
    value:   2
```

Listing 5.11: Even though this function is correct, Prusti cannot prove that the postcondition holds. Prusti provides a counterexample that seems incorrect compared to the actual Rust code, but compared to the given specification of the functions `new()` and `deref()` it is actually correct.

terexample from Listing 5.13. As expected, Prusti was not able to produce a meaningful value of the argument `t`. Nevertheless, it produced values for the functions `height()` and `number_of_nodes()`. This information is enough to identify the structure of the binary tree, in this case a binary tree with exactly one branch.

Of course having a complete counterexample would be preferable. Nevertheless, this is only temporarily and will be resolved once the Rust `box` type is fully implemented. After that, Prusti will be able to produce a value for the binary tree as well.

```rust
struct BinaryTree<i8> {
    val: i8,
    left: Option<BoxWrapper<BinaryTree<i8>>>,
    right: Option<BoxWrapper<BinaryTree<i8>>>,
}

impl BinaryTree<i8> {
    #[pure]
    #[ensures(result >= Int::new(1))]
    fn height(&self) -> Int { ... }
    #[pure]
    #[ensures(result >= Int::new(1))]
    fn number_of_nodes(&self) -> Int{ ... }
}

#[requires(t.height() == Int::new(3))]
fn unbalanced(t: BinaryTree<i8>){
    let b = t.height() != t.number_of_nodes()
    assert!(b)
}
```

Listing 5.12: In the definition of the `BinaryTree` the `box` type is replaced by its wrapper. The functions in the implementation describe two properties of the binary tree, the height and the number of nodes. We assume that the reader is familiar with binary trees and omit the actual implementation of those functions. The function `unbalanced` checks whether the height and the number of nodes are not equal. The precondition is only used to force a specific counterexample and could be removed.

```
the asserted expression might not hold
counterexample for "t"
    value:  BinaryTree {
              val: 1,
              left: Some(BoxWrapper(?)),
              right: None,
            }
counterexample for "height"
    value:  3
counterexample for "number_of_nodes"
    value:  3
```

Listing 5.13: Counterexample provided by the refactored version of Prusti for the function `unbalanced()`.

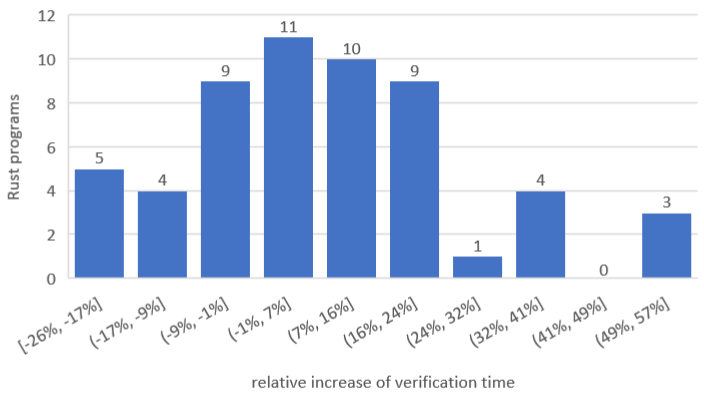


Figure 5.1: Histogram of relative increase of verification time.

5.3 Timing Analysis

The impact of the counterexample generation algorithm on the refactored version of Prusti's overall performance was evaluated using a slightly modified version of Prusti's benchmarking script. It runs with 56 Rust programs, once with Prusti's counterexample generation enabled and once with it disabled. Each run verified each Rust program ten times and averaged the verification time.

The test programs contain a variety of different types, different number of variable and different number of functions. Throughout testing we discovered that verification of Rust programs containing loops tends to take twice as long with counterexample generation enabled. For that reason we excluded loops from this analysis.

The result of this timing analysis showed that counterexamples increase verification time on average by 10%. A histogram of all relative impacts are shown in Figure 5.1.

While the increase of verification time is below 10% for 75% of all tested programs, there are a few outliers. In extreme cases the counterexample generation increased verification time by 50%. A closer inspection of these extreme cases revealed that the programs either create large Viper programs or contain a large number of functions. This suggests that the variable mapping is currently the performance bottleneck.

Since the counterexample generation feature is meant to be used for debugging and is not expected to be enabled permanently, the overall performance loss seems tolerable.

Chapter 6

Conclusion

In this thesis we have improved the counterexample support of Prusti. This was achieved by using a different type encoding. Our work has mainly focused on the refactored version and on how its different implementation could be leveraged to generate better counterexamples.

For that purpose we have implemented a new algorithm which translates functions from Silicon's counterexample back to a level of Rust and reports this in a customizable way back to the user.

As we saw in the evaluation, the new algorithm improves the counterexample generation in the refactored version to a great extent, but only provides little improvement in the old version. This is due to two limiting factors: either the snapshot-based type encoding is not created in the first place, or it is created but not found by the algorithm as described in Section 4.2.1. However, the snapshot-based type encoding and the refactored version with its features, e.g. SSA, are a very much more suitable combination for a counterexample generation. It allows Prusti to create much more fine-grained and detailed counterexamples than before.

In terms of complex data structures, the evaluation shows that the refactored version of Prusti is now capable of producing counterexamples for complex types in many cases. It is mainly limited by the fact that the box type is not fully implemented. Once this is done, we expect that the counterexample generation will work even better with complex data structures.

Since counterexamples are now more detailed, we gave the user the possibility to customize the output of the counterexample. This is especially helpful for complex data structures.

6.1 Future Work

In this section we present further possible improvements for Prusti's counterexample generation:

- Prusti's counterexample algorithm could be extended to support additional types, e.g. maps or pointers.
- Prusti is commonly used in VSCode with the extension Prusti Assistant [5]. The counterexamples are presented as clickable messages. If those messages are clicked on, the cursor automatically jumps to the corresponding line of code. This could be extended in such a way that the counterexample is directly inlined within the code.
- The current output of a counterexample is a simple message which contains a list with all concrete values per variable. The information gathered during the translation process could be used to generate a control-flow graph of the Rust program and showcase which path was taken by the counterexample. This would simplify the user's debugging process even further.
- We provided a macro to the user to customize the output of a counterexample. At the moment the user is only allowed to inline field values of structs within a String. In the future Prusti could allow arbitrary functions instead of just fields. This would give the user much more freedom in changing the output of a counterexample.
- In general, we assume a more detailed counterexample is better than a less detailed one. However, if programs get larger and use more variables there might be too much irrelevant information in the counterexample. In cases like this, Prusti could provide an option for the user to omit certain parts of the counterexample, or alternatively highlight certain parts.

Bibliography

[1] C/C++ reference. https://en.cppreference.com/w/. Online. Accessed on 2022-08-14.

[2] Heap-based type encoding. https://viperproject.github.io/prusti-dev/dev-guide/encoding/types-heap.html. Online. Accessed on 2022-08-14.

[3] Mid-level intermediate representation. https://rustc-dev-guide.rust-lang.org/mir/index.html. Online. Accessed on 2022-08-14.

[4] Models. https://viperproject.github.io/prusti-dev/user-guide/verify/type-models.html. Online. Accessed on 2022-08-14.

[5] Prusti assistant. https://github.com/viperproject/prusti-assistant. Online. Accessed on 2022-08-14.

[6] Rust programming language. https://www.rust-lang.org/. Online. Accessed on 2022-08-14.

[7] Snapshot-based type encoding. https://viperproject.github.io/prusti-dev/dev-guide/encoding/types-snap.html. Online. Accessed on 2022-08-14.

[8] Viper tutorial. http://viper.ethz.ch/tutorial/. Online. Accessed on 2022-08-14.

[9] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. Technical report, ETH Zurich, 2019.

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

[11] Microsoft. Z3 prover. https://github.com/Z3Prover/. Online. Accessed on 2022-08-14.

[12] 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.

[13] M. Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. PhD thesis, ETH Zurich, 2016.

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):

Counterexamples for Complex Data Structures for 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):**
Limbeck	Markus

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)**
Zurich, 14.08.2022	Markus Limbeck

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