Counterexamples for a Rust Verifier

Bachelor Thesis

Cedric Hegglin

May 5, 2021

Advisors: Prof. Dr. Peter Müller, Dr. Christoph Matheja, Aurel Bílý

Department of Computer Science, ETH Zürich

Abstract

The Rust programming language is a more secure alternative to traditional systems programming languages like C or C++. Prusti is a deductive verifier for Rust aiming to make verification lightweight and accessible. It allows programmers to annotate Rust programs with specifications and prove their functional correctness. However, in case of verification failures, the provided information is very limited, and figuring out the reason for an error can be challenging. Counterexamples can facilitate the debugging of a program and its specification by providing concrete values that cause an error. The main contribution of this thesis is an extension of Prusti, that generates meaningful counterexamples if verification fails and presents them similar to Rust compiler errors. The counterexamples are obtained by leveraging existing features of the Viper verification infrastructure underlying Prusti. Specifically, we translate the counterexamples generated by Viper's back-end Silicon. As part of this project, we also contributed an improved counterexample representation to Silicon.

Acknowledgements

Contents

Chapter 1

Introduction

Software is all around us and, as our dependency on it is constantly growing, ensuring its correctness is becoming ever more important. Unfortunately, especially in systems programming writing correct programs is notoriously difficult. The Rust programming language [1] is an alternative to languages like C and C++ that prevents programmers from introducing some of the most common errors such as data races by enforcing a strict typing discipline and ownership model.

Prusti [3] is a deductive verifier for the Rust programming language, extending Rust's existing safety properties to functional correctness. It leverages Rust's type system to allow users to annotate programs with specifications that are mostly using Rust syntax while hiding the underlying complexity. When verifying a Rust program using Prusti it tries to prove that each function can never reach an error state and that it always satisfies its specifications. An example of an annotated function with a post-condition is shown in Listing 1.1.

Listing 1.1: Example of a Rust program with Prusti annotations

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

For positive n this function computes $\sum_{i=1}^{n} i$, which is equal to $\frac{n*(n+1)}{2}$, but for negative inputs it simply returns zero. Running Prusti[1] on this function will fail verification with the following message:

```
error: [Prusti: verification error] postcondition might not hold.
  --> sum.rs:5:11
   |
5  | #[ensures(result == n*(n+1)/2)]
   |           ^^^^^^^^^^^^^^^^^^^^
```

The verification fails because the post-condition of this function does not hold for a negative argument. However, if the user is unaware of the reason for the failure, Prusti's error message does not help him with figuring out how this violation can be reached. Rather than manually figuring out why the post-condition might not hold, it would be better if Prusti gave us debugging hints in the form of counterexamples. A counterexample for our previous function sum would be {n <- -1, result <- 0}, i.e. an assignment of the variables leading to a violation. Given this information, figuring out the cause of the verification error is straightforward. The given example is rather simple and counterexamples become even more useful for more complicated code.

The main goal of this thesis project is to add support for counterexamples in Prusti to facilitate debugging of Rust programs and their specifications. Our implementation of counterexamples exploits existing features of the Viper verification infrastructure [8] underlying Prusti. Specifically, Prusti translates Rust programs to the Viper intermediate verification language and verifies them with Silicon, one of Viper's back-ends. While Silicon already supports counterexample generation, Prusti initially did not take advantage of this capability. The core idea for this project is to use Silicon's counterexamples in Prusti and to translate them to counterexamples for a Rust program.

However, Silicon's counterexamples initially contained various Viper internal representations and its information was spread over various data structures. Therefore, part of the effort of this project went into making Silicon's representation of counterexamples more accessible and compact.

The main contributions of this project are the following:

- We improve the counterexample support of Silicon for both direct users and front-ends by adding a simplified representation.

- We extended Prusti to make use of Silicon's new representation to generate counterexamples for Rust programs.

- We discuss several ways of presenting counterexamples.

---

[1]as of commit 9546a21

- We implement one way of presenting Rust counterexamples for Prusti.

- We discuss the current limitations of our implementation and their source and explore some approaches on how to resolve them.

In the following chapter, we provide some information on Prusti, Silicon, and counterexamples in general. Chapter 3 presents the high level idea of our approach. Chapter 4 gives details on the implementation of both the improved Silicon counterexamples and Prusti's new counterexamples. In Chapter 5, we evaluate our Rust counterexamples and showcase their usefulness as well as their limitations. Finally, Chapter 6 summarizes our findings and delves into some ideas for improvements and continuations of our project in the future.

For the remainder of this thesis, the reader is assumed to have some basic understanding of the Viper verification language (references, permissions and predicates) [15] and the Rust programming language.

Chapter 2

Background

In this chapter, we give some necessary backgrounds on counterexamples and the relevant parts of Prusti and Silicon. In particular, we look at two existing implementations of counterexamples, Prusti's encoding from Rust to Viper and discuss Silicon's[1] generated counterexamples.

2.1 Counterexamples

When verifying a function, it can be challenging to determine the source of a verification error. If we run Prusti [3] in its state before our project[2] on the function given in Listing 2.1, it only reports that its post-condition might not hold as shown in Listing 2.2. The function computes the maximum of two values and we try to prove that its result has to be larger than one of the two values. Prusti fails to verify this function because it violates the post-condition if the two values are equal. While it is easy to figure out why the verification fails in this case, finding the source of an error becomes more involved for larger, realistic programs.

Listing 2.1: Example Rust function with Prusti annotations

```
#[ensures(result > x || result > y)]
fn max(x: i32, y: i32) -> i32 {
    if x > y {
        x
    } else {
        y
    }
}
```

---

[1]as of commit f930cf1

[2]as of commit 9546a21, i.e. before the implementation of counterexamples

Listing 2.2: Output generated by Prusti in its initial state

```
error: [Prusti: verification error] postcondition might not hold.
 --> max.rs:4:11
  |
4 | #[ensures(result > x || result > y)]
  |           ^^^^^^^^^^^^^^^^^^^^^^^^^
  |
...
Verification failed
error: aborting due to previous error; 1 warning emitted
```

Ideally, Prusti should not only tell us that verification failed but justify the failure with a counterexample. As already mentioned in Chapter 1, in the context of SMT-based verifiers, counterexamples usually provide a model containing the values of variables at various program points. We speak of a *valid counterexample* if it does not violate any pre-condition, matches the values of an actual execution of the program and leads to some error state. If verification fails to prove the correctness of a certain program, this can give us valuable information why it is failing. For the previous example {x:2, y:2, result:2} is a valid counterexample because there are no pre-conditions, executing the function with these arguments will always lead to the given result and the result does violate the post-condition. {x:1, y:2, result:2} is an invalid counterexample because even though it represents a correct execution of the function it does not violate the post-condition.

To give an idea of different ways to present counterexamples we look at two verifiers that already implement counterexamples. One of those verifiers is Nagini [6], another Viper frontend but for the Python programming language. If we run Nagini on the analogous Python function of our previously considered example, we get the result presented in Listing 2.3. It provides the values of the arguments when entering the function and the values of all

Listing 2.3: Counterexample given by Nagini

```
Old store:
  x -> False,
  y -> 0
Old heap: Empty.
Current store:
  x -> False,
  y -> 0,
  Result() -> 0
Current heap: Empty.
```

local variables when the function returns. The value `False` for x might be confusing, but it stems from the fact that booleans are a subtype of integers in Python and represent zero. Therefore this is a valid counterexample.

Another widely used verifier is Dafny [7]. When using Dafny with its Visual Studio Code (VSCode) extension, it displays the values of the counterexample directly within the code (in blue). It shows the values of the local variables at the various points during execution and therefore also displays which paths are taken. Its result for the same function as before is shown in Figure 2.1. Additionally, even though it is not displayed in the given example, Dafny also displays the values of variables at each point where their values change and even displays the return values of function calls within the method. This can be crucial in some cases, for example, if there is a call to a nondeterministic function like a random number generator and the verification error depends on its return value. It can potentially be very hard to reproduce or understand the error without this information.

```
method Max(x:int, y:int) returns (n:int)
    ensures (n > x || n > y)
    {    x = 0; y = 0;
        if (x>y){
            n := x;
        } else {
            n := y;    x = 0; y = 0; n = 0;
        }
    }
```

Figure 2.1: Method annotated with Dafny's counterexample

On a more general note on counterexamples in the context of symbolic execution, we should know they can be *spurious*. Spurious counterexamples can occur due to the abstractions performed during symbolic execution since it is usually sound but not complete. They produce a variable assignment proposing that the function reached some error state, but this state is not reachable for a concrete execution of the function. An example where we would expect this behavior is shown in Listing 2.4, where the function use_inc will always satisfy its post-condition. However, because of Prusti's modular verification and the weak specification for the function `incr`, the verifier over-approximates the set of reachable states and fails verification. Therefore we expect a spurious counterexample value for the function's result, in this case meaning a value not equal to 20 and greater than 15.

Listing 2.4: Example for expected spurious behavior

```rust
#[ensures(result > x + 5)]
fn incr(x: i32) -> i32 {
    x + 10
}


#[ensures(result == 20)]
fn use_inc() -> i32 {
    let x = 10;
    incr(x)
}
```

2.2 Verification Pipeline

To be able to understand the approach and implementation of this project, a basic understanding of Prusti and Silicon in their state before our contribution is necessary. In this section, we will cover the relevant parts of Prusti's verification process and Silicon's available but unused counterexamples by going through various parts of the verification process for our previous example from Listing 2.1.

It is important to know that Prusti [3] is an extension of the Rust compiler and has access to its internals. Given a Rust program, Prusti invokes the Rust compiler on this program and mostly works with the mid-level intermediate representation (MIR) of the program. The MIR encoding contains an explicit control flow graph with a label for each basic block. In this representation the result of a function is always stored in the variable _0 and the arguments are the subsequent variables (_1 and _2 in our example). Prusti then proceeds to translate the generated MIR encoding to a Viper program. This translation is quite complex but in the context of this project, we are mostly focusing on the encoding of various types and the generated Viper labels. Chapter 4 will delve more into this translation, but the resulting arguments of the generated program for our considered example will have the following structure:

```
_1 <- Ref {
    val_int: Int
}
_2 <- Ref {
    val_int: Int
}
```

Additionally, we need to know that Prusti generates various labels in the translated Viper program, according to the control flow graph of the generated MIR encoding. After the translation, Prusti runs Silicon to verify the generated Viper program and uses its result. If the verification fails, Silicon's errors are mapped to their origin in the Rust program. In this case, Silicon is able to produce a counterexample but Prusti did not take advantage of that capability. Since this project heavily relies on the counterexamples provided by Silicon, we need to understand them in their state before our contribution. Their implementation is the result of a previous thesis called "SMT Models for Verification Debugging" [10] and in the following, we will have a first look at what information they provide.

In the initial state of Silicon, it generates a so-called native model and a state for each method that fails verification. The native model is by the SMT solver. It is a map of identifiers to the model's entries which can be values of variables at certain points in the program or results of called functions among other things. The state contains various data structures of which the following are relevant for a counterexample:

- the store, a map from the names of local variables to their value or another identifier pointing to some location in the other data structures;

- the main heap, containing so-called chunks of predicates and fields of references representing their values when the method fails verification; and

- the old heaps, a map from labels to heaps, where each heap has the same structure as the main heap, but it represents all the values of the method's variables when crossing the corresponding label. It always contains a heap at the label "old", representing the values when entering the method, and additional heaps if the execution runs through explicitly defined labels, e.g. the ones generated for basic blocks by Prusti.

To get an idea of how we can use these counterexamples but also to emphasize how cumbersome it was to use them in their initial state, we will look at some concrete examples. First, we consider a heap-independent Viper method (Listing 2.5) that tries to prove the same thing as our previous example. A valid counterexample would give us three equal values for both of the arguments and the result. If we look at the relevant parts of the generated counterexample (Listing 2.6) it is fairly easy to extract this information by mapping the provided identifiers from the store to the given model entries, giving us a value of 906 for all the variables.

Listing 2.5: Heap independent Viper example

```
method max(x: Int, y: Int) returns (n:Int)
    ensures(n > x || n > y)
    {
        if (x > y) {
            n := x
        } else {
            n := y
        }
    }
```

Listing 2.6: Relevant parts of counterexample for Listing 2.5

```
Store {
    x -> x@3@04,
    y -> y@4@04,
    n -> y@4@04,
}

Model {
    x@3@04 -> 906,
    y@4@04 -> 906,
}
```

Unfortunately, extracting this information becomes a lot more complicated once we start dealing with values involving the heap. To demonstrate the problem we will have a look at a simplified version of the method that is generated by Prusti for our initial example (Listing 2.7). We do not use Prusti's generated Viper code because it is hard to read and the produced files are a lot larger. The fields of the arguments are now located on the heap and hence, the resulting counterexample suddenly involves a lot of Viper internal structures. Extracting meaningful information, even with the required knowledge, is tedious. If we wanted to get the value of _1, we would first have to get the identifier _1@3@04 from the store (line 2 in Listing 2.8). On the heap, we can then find the location of the field for _1.val_int on line 8, given by the SortWrapper construct. We do not have to understand this term completely yet, but it points us to the matching field of the model (line 12) where we can get a value of 323 for this field. This is still a fairly simple example, but the point to be made here is that the current interface for counterexamples in Silicon is hard to use for both direct users and front-ends of Silicon.

Listing 2.7: Viper program similar to the one generated by Prusti

```
field val_int: Int
method max(_1:Ref, _2:Ref) returns (_0:Int)
    requires(acc(_1.val_int) && acc(_2.val_int))
    ensures(_0 > old(_1.val_int) || _0 > old(_2.val_int))
    {
        if (_1.val_int > _2.val_int) {
            _0 := _1.val_int
        } else {
            _0 := _2.val_int
        }
    }
```

Listing 2.8: relevant parts of counterexample for Listing 2.7

```
1   Store {
2       _1 -> _1@3@04,
3       _2 -> _2@4@04,
4       _0 -> _0@9@04,
5   },
6   Heap {
7       _2@4@04.val_int -> SortWrapper.SnapToInt(Second(t@6@04)),
8       _1@3@04.val_int -> SortWrapper.SnapToInt(First(t@6@04)),
9   },
10  Model {
11      t@6@04 -> (Snap.combine Snap.unit Snap.unit),
12      SortWrappers.SnapToInt -> {
13          323
14      },
15      _0@9@04 -> 323,
16  }
```

Chapter 3

Approach

As outlined in Chapter 1, the idea of this project is to use the counterexamples generated by Silicon to generate meaningful counterexamples for Rust programs. On a high level, this goal can be split into three parts: First, extracting the needed information from Silicon's counterexamples; Second, translating this information back to their corresponding Rust components; and finally, presenting the counterexamples to the user in an understand-
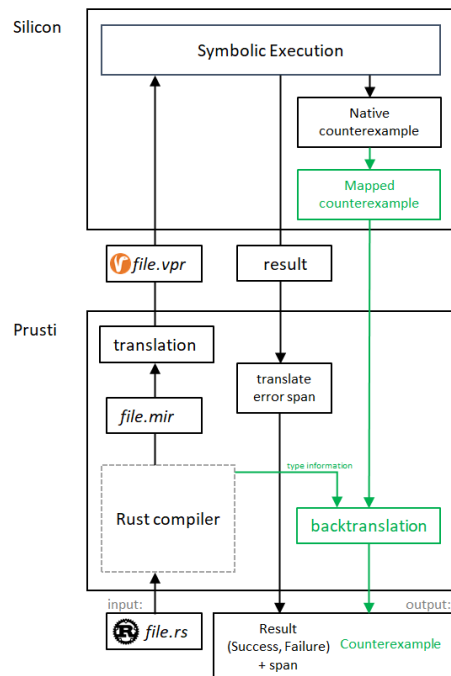

Figure 3.1: Prusti's structure with new components (green).

able way. Figure 3.1 illustrates Prusti's [3] structure and the components we contribute (in green).

As we have seen in Section 2.2, the counterexamples that were initially available in Silicon are difficult to work with and if we are interested in the values of fields of references, there is a lot of processing required. Since Prusti translates almost every Rust type to a reference in Viper, we added a simpler representation of counterexamples to Silicon that contains an entry for each variable. Such an entry recursively contains other entries for its fields if it is a reference. To obtain this new representation it evaluates all Silicon internal values found on the heap and within the model that we have seen previously and maps the given values of the fields to the corresponding references. This is also done with the heaps at all the old labels that are available, giving us multiple values for variables that are mutated during execution. While all of this could be implemented within Prusti, we decided to do it on the Silicon level and design it in a way that makes it front-end independent. This has the advantage that it might be used in future implementations of counterexamples for other front-ends but can also be useful to Viper users in general.

To give an idea of what our new representation achieves, we take a look at another version of our maximum function shown in Listing 3.1, computing the maximum of two values with the slight difference that the permissions to the fields are inhaled in the method's body instead of giving them as a precondition. Listing 3.2 shows us the new representation of the counterexample that is generated for the example. Looking at the values given after the method returns, we can see the advantage over the initial native counterexamples. However, it is worth noting that at the label "old", where the permission to the fields is not inhaled yet, we do not get any values for the fields of the references.

Listing 3.1: Adjusted Viper example

```
field val_int: Int
method max(_1:Ref, _2:Ref) returns (_0:Int)
    ensures(acc(_1.val_int) && acc(_2.val_int))
    ensures(_0 > _1.val_int || _0 > _2.val_int)
    {
        inhale(acc(_1.val_int) && acc(_2.val_int))
        if (_1.val_int > _2.val_int) {
            _0 := _1.val_int
        } else {
            _0 := _2.val_int
        }
    }
```

Listing 3.2: Mapped counterexample for Listing 3.1

```
model at label: old
_1 <- Ref ($Ref!val!0) {
}
_2 <- Ref ($Ref!val!2) {
}
_0 <- 906

on return:
_1 <- Ref ($Ref!val!0) {
        val_int(perm: 1/1) <- 906
}
_2 <- Ref ($Ref!val!2) {
        val_int(perm: 1/1) <- 906
}
_0 <- 906
```

On the Prusti side, we use Silicon's new representation combined with the compiler's information to obtain counterexamples for Rust. We first need to find a mapping of all Rust variables of the failing function to their corresponding variable names in Viper. Then we try to get a counterexample entry for each Viper variable and revert the translation of the various types that Prusti performed according to the type of the Rust variable which we get from the compiler. We try to translate every variable according to its type if the necessary information is available, which needs to be done recursively for composite types like tuples, structs or references. For example, if we assume that _1 in the given counterexample is the entry for x back in our example Listing 2.1, the compiler would tell us that x is an integer and we would get its value from _1.val_int. This sort of back-translation is supported for integers, booleans, characters, references, tuples, structs, and enums.

The last part of our approach is presenting the counterexamples to the user. Before starting any implementation we studied various Rust examples that fail verification to understand how to meaningfully present counterexamples. One way of presenting counterexamples is to give the initial value of all arguments, and all the values of the local variables including the result and the arguments once the function fails. In almost all considered cases this makes determining the reason for the verification error straightforward. That is why, for any failing function, Prusti now computes the available values at those two points. They are presented to the user as a compiler note for each variable.

Chapter 4

Implementation

In this chapter, we will delve into the details of the implementation of our project. It will be split into two parts, first explaining the changes we applied to Silicon to obtain a simpler, more compact representation of counterexamples and second explaining how this new representation can be used in Prusti to generate counterexamples for Rust programs.

4.1 Improving Counterexamples in Silicon

In this section, we go into more detail on the implementation of the new mapped Silicon counterexamples. Some parts of the implementation are based on Nagini's [6] implementation, which is the first Viper front-end that implemented counterexamples. With the goal of making the available information more usable, we implemented a new class called `MappedCounterexample` and a class `Converter` that is responsible for translating the existing information to the new representation. The new counterexamples are generated if the user executes Silicon with the flag "`--counterexample mapped`".

4.1.1 Native Models

First, we need to have a deeper look at the initially available native model [14] that we briefly discussed in Chapter 2. While the heap and store are also crucially important to process the model correctly, all of the concrete values of a counterexample that tell us when a method fails are found in the model. The native model is a map of strings to instances of a class called `ModelEntry`. The structure of this class and its sub-classes is shown in Figure 4.1.

Instances of the class `ConstantEntry` represent, as the name suggests, constant values given as strings. These can be the strings of integer or boolean
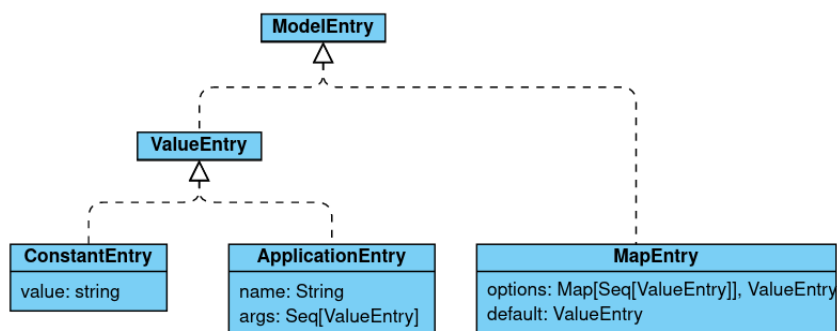
Figure 4.1: UML diagram of the `ModelEntry` class

values, strings of identifiers (e.g. for references), or the string of internal values such as `Snap.unit`. The class `ApplicationEntry`, which is also part of the class `ValueEntry`, contains the name of a function and a sequence of arguments. The following listing contains examples of `ApplicationEntry` instances we typically encounter in generated models:

```
$t@2@04 -> ($Snap.combine $Snap.unit $Snap.unit)
x@1@04 -> (- 563)
```

In the printed output of counterexamples, shown in the above listing, instances of `ApplicationEntry` are always displayed as S-expressions, meaning that they are put into parentheses containing their function name followed by the arguments. Each arguments is either a `ConstantEntry` or another `ApplicationEntry`, allowing for a nested structure.

Instances of the class `MapEntry` represent the result of function calls and define a mapping from a sequence of `ValueEntry`s to a single `ValueEntry`. They store results of function calls during the execution and help to recover those results for given values of the arguments. Those entries are not only generated for functions that are explicitly defined in the program we are verifying, but also for Silicon internal functions like sort wrappers. To get a result for a given sequence of arguments we try to find the arguments in the map `options` (Figure 4.1) and use the `ValueEntry` it maps to if we find it. Otherwise, we take the default value, which is shown as the `else` case in the printed output. In the printed version of the model `MapEntry` instances are always surrounded by curly brackets as we can see in the following listing:

```
SortWrappers.SnapToInt -> {
    (Snap.combine Snap.unit Snap.unit) -> 21
    else -> 42
}
```

4.1.2 ExtractedModel

Before looking into the translation and processing of the heap and state using the native model, we need to know what our desired result will look like. The new counterexample will consist of several models containing the values of local variables at each label of a method. Each model is a map from identifiers of variables to a new class of entries. The new class is called `ExtractedModelEntry` and contains sub-classes for the various values we are interested in for a meaningful counterexample. Listing 4.1 shows the various types of entries our new model will contain. The first three classes represent values of Viper's basic types. Then there are three classes to represent references. In Silicon, every instance of a reference has a unique identifier of the form `Ref!Val!0` which is stored in the name field of these entries. The class `RefEntry` also contains all of its fields as a mapping from the field name to another entry and the permission to that field. The class

Listing 4.1: Structure of `ExtractedModelEntry` (Scala)

```scala
sealed trait ExtractedModelEntry

//Literals
case class LitIntEntry(value: BigInt)
case class LitBoolEntry(value: Boolean)
case class LitPermEntry(value: Double)

//References
case class RefEntry(
    name: String,
    fields: Map[String, (ExtractedModelEntry, Permission)]],
)
case class NullRefEntry(name: String)
case class RecursiveRefEntry(name: String)

//Sequences
case class SeqEntry(
    name: String,
    values: Vector[ExtractedModelEntry]
}

//Debugging and intermediate results
case class VarEntry(value: String, sort: Sort)
case class OtherEntry(value: String, problem: String)
case class UnprocessedModelEntry(entry: ValueEntry)
```

`RecursiveRefEntry` was added to avoid infinite recursion for cyclic references. We also added support for sequences while other composite types like sets and multi-sets remain as future work. Finally, there are a few types of entries that either store some intermediate result or help reporting the reason why a value of an entry could not be extracted. For example, instances of the class `VarEntry` store identifiers of sequences and references that will be replaced with their corresponding entries during further processing.

4.1.3 Evaluating Terms

Both on the heap and in the store, we will encounter instances of Silicon's `Term` class [14]. To get a meaningful counterexample we need to be able to evaluate those terms for a given native model. We implemented a function `evaluateTerm` which takes a native model and a term as arguments and returns an `ExtractedModelEntry`. Its implementation is based on Nagini [6] [5], with the difference that Nagini's version operates on an older version of the native model and that its method only returns strings. The function supports a subset of Silicon's terms and in the following, we are going to explain how the most frequently occurring terms are evaluated.

Variables

Variable terms are of the form `Var(name, sort)` where the sort defines their type. To evaluate them, we always try to find an entry in the native model mapped to their name. If we do not find one we return an `OtherEntry` reporting that it cannot be found. Otherwise, what we do to evaluate the found entry depends on the sort of the variable; there are three cases:

- If the sort is `Int`, `Bool` or `Perm`, we try to get an actual value from the native entry and return a literal entry. The found entry either has to be a `ConstantEntry` or an `ApplicationEntry` of the function "-" containing another constant, meaning that the contained value has to be negated. The `ConstantEntry` can be parsed to a value of the required type. The function, if there is one, has to be applied to get the final value. We return one of the three literal entries depending on the sort. Note that the sort `Perm` is not only used for fractional permissions but also for the type `Rational` in Viper. The following listing contains three examples of native entries for each of the discussed sorts:

```
x@0@04 -> 1.0,
y@1@04 -> False,
z@2@04 -> (- 7)
```

- If the sort is `Ref` or `Sequence`, then the found entry needs to be a `ConstantEntry` containing the string of an identifier. The evaluation

returns a `VarEntry` containing the found identifier and its sort. We are interested in the values of the sequence or the fields of the reference, but they have to be resolved at a later stage and the `VarEntry` serves as a placeholder. The following listing contains native model entries generated for a sequence and a reference:

```
x@1@04 -> Seq<Int>!val!1,
y@4@04 -> $Ref!val!2
```

- If the sort is `Snap`, we return an `UnprocessedModelEntry` directly containing the found native entry. Why this is needed will become clear when we evaluate the next term.

First/Second

The terms `First` and `Second` are part of Silicon's modelling of the heap. They can only be applied to `ApplicationEntry` instances of the form `(Snap.combine x y)` where `First` returns x and `Second` returns y. They contain another term that, when evaluated recursively, should return an `UnprocessedModelEntry` containing the entry to which we can apply them. To give an example, we evaluate `Second(t@4@04)` with the following model entry:

```
t@4@04 -> (Snap.combine Snap.unit (Snap.combine Snap.unit Snap.unit))
```

We first recursively evaluate the variable `t@4@04`, returning an `Unprocessed-ModelEntry` containing the entry shown in the above listing. We then apply the function second to the contained entry, giving us the `ApplicationEntry` `(Snap.combine Snap.unit Snap.unit)` which we return in another `UnprocessedModelEntry`.

Sort Wrappers

The last terms we are covering are the sort wrappers. They contain two sorts and another term. The two sorts are needed to construct the name of the `MapEntry` it refers to in the model, e.g. SnapToRef. To evaluate a term, for example, `SortWrapper.SnapToRef(First(Second(t@4@04)))`, we first evaluate the contained term recursively. It should return another `Unprocessed-ModelEntry` containing an `ApplicationEntry`, for example `(Snap.combine Snap.unit Snap.unit)`. In the native model we should find a `MapEntry` of the following form:

```
SortWrapper.SnapToRef -> {
    (Snap.combine Snap.unit Snap.unit) -> Ref!val!4
    else -> Ref!val!3
}
```

The returned value of our recursive evaluation maps to `Ref!val!4`, and therefore we return this identifier in a `VarEntry`. Depending on the sort we expect, the result could also be one of the literal entries.

4.1.4 Heap Extraction

With the new function `evaluateTerm`, we can now start processing the heaps at the various labels. A heap in Silicon is a list of chunks. There are multiple types of chunks in Silicon but we only handle so-called basic chunks which are either predicate or field chunks. This is another design decision we reproduced from Nagini [6] and they seem to contain all the relevant information for the supported features. A field chunk consists of an argument term, a field name, and a snap term. It tells us that the value of `arg.fieldname` is given by the snap term. For example, if we run Silicon on the method in Listing 4.3, it generates the main heap and the heap at the label "old". Since no values change during the execution of this method, we only need to look at one of the heaps, which is shown in the following:

```
SortWrapper.SnapToRef(First(t@2@04).field1 ->
    SortWrapper.SnapToInt(Second(Second(t@2@04)),
x@1@04.next -> SortWrapper.SnapToRef(First($t@2@04)),
SortWrapper.SnapToRef(First(t@2@04).field2 ->
    SortWrapper.SnapToInt(First(Second(t@2@04)),
```

The goal of the extraction is to generate an expression of the form `identifier.fieldname = entry` for each field chunk. We can get that information by using our previously discussed function, `evaluateTerm`, on both the argument and the snap term. The argument has to evaluate to a `VarEntry` and we only keep its name. For the above example the result for the extraction, given the native model in Listing 4.4, is the following:

Listing 4.2: Extracted heap

```
Ref!val!2.field1 <- 1,
Ref!val!0.next <- Ref!val!2,
Ref!val!2.field2 <- 0
```

All the extracted information is stored in a new `ExtractedHeap` class and will be used for the final mapping as explained in the next section. While predicate chunks might also be useful for some applications, we currently are not using them for our counterexamples in Prusti. They are also processed and part of the extracted heap but for the final mapping only field chunks are relevant, which is why we are not covering them in more detail.

Listing 4.3: Example Viper program with nested references

```
field field1: Int
field field2: Int
field next: Ref

method simple(x: Ref)
    requires acc(x.next)
    requires acc(x.next.field2) && acc(x.next.field1)
    {
        assert(x.next.field2 == x.next.field1)
    }
```

Listing 4.4: Native counterexample for Listing 4.3

```
x@1@04 -> Ref!val!0,
t@1@04 -> (Snap.combine
    Snap.unit
    (Snap.combine
        Snap.unit
        (Snap.combine Snap.unit Snap.unit)
    )
),
SortWrappers.SnapToInt -> {
    (Snap.combine Snap.unit Snap.unit) -> 1
    else -> 0
},
SortWrappers.SnapToRef -> {
    Ref!val!2
},
```

4.1.5 Final Mapping

After processing all the heaps we can now try to get the actual values of a counterexample. We evaluate the store at each of the given heaps to get the values of the method at those labels. Since no values change during the execution of the previous example, we will again look at one heap only. Now our goal is to get a counterexample value for each variable that is located in the store. The store is a mapping of variables to terms and for our previous example in Listing 4.3 it only contains the value x -> x@1@04.

If we want to get an entry for the variable x, we start by using our eval-uateTerm function on x@1@04 with the model given in Listing 4.4. In some cases the result might be one of the literal entries, meaning a value for the variable. However, since it is a reference we get a VarEntry containing the identifier Ref!val!0. We want to turn it into a RefEntry containing all the

fields, so we start going through the previously extracted heap in Listing 4.2 and look for fields of the current identifier. We find one extracted chunk for the identifier `Ref!val!0`, storing the value of the field `next` given by another `VarEntry` containing `Ref!val!2`. Because it is another `VarEntry` we need to resolve it recursively. For the reference `Ref!val!2` we find two fields containing the literal integer-entries for 0 and 1 and we finally obtain the following `ExtractedModelEntry` for the value of x:

```
x <- Ref (Ref!val!0) {
    next <- Ref (Ref!val!2) {
        field1 <- 1
        field2 <- 0
    }
}
```

While going through the fields of a nested reference, we keep a list of identifiers we already encountered. If we find a reference for a second time, instead of mapping its fields again, we return a `RecursiveRefEntry` to avoid infinite recursion if there are cyclic references. Finally, if we encounter a `VarEntry` of a sequence, we also resolve it to a `SequenceEntry` during the final mapping. If a counterexample involves a sequence, we will find the following two `MapEntry` instances in the model:

```
Seq_length -> {
    Seq<Int>!val!1 -> 5
    else -> 0
},
Seq_index -> {
    Seq<Int>!val!1 1 -> 4
    Seq<Int>!val!1 2 -> 6
    else -> 3
}
```

Given the identifier from the variable-entry, in this case `Seq<Int>!val!1`, we can first get the length of the sequence from the `Seq_length` field. Then we can get the value at each index by evaluating the `Seq_index` function given the identifier and index as the arguments. We get a sequence of length 5 with the elements `[3, 4, 6, 3, 3]`.

Recall that we always have a main heap and a list of heaps for all the labels of the Viper program. After the mapping, our result is a main model, giving us the values of the method when the verification error occurred, and a map

from labels to older models. The map always contains a label "old", giving us the model for the values of variables when entering the method, and additional labels if the method contains explicitly defined labels.

4.2 Prusti Counterexamples

In this section, we will explain how Silicon's new representation of counterexamples can be used to generate counterexamples for Rust programs. To enable counterexample generation in our extension of Prusti, one has to set the environment variable `PRUSTI_COUNTEREXAMPLE` to true. In that case, Prusti passes the additional flag "`--counterexample mapped`" to Silicon. If Silicon's verification fails for a function, we should be able to get a Silicon counterexample for it. The produced counterexample is then copied over into Rust to a data structure equivalent to Silicon's mapped counterexample. After we encounter a verification error and obtain a Silicon counterexample, we proceed with the following simplified steps: we identify the functions that fail verification, for each function we determine a set of Rust variables we are interested in, for each variable we look for its corresponding entry in the Silicon counterexample and translate it back to a Rust value, and finally, we present the counterexample to the user. These steps will be explained in more detail in the following sections.

4.2.1 Identifying the failing function

Let us assume we are given an annotated Rust function that Prusti failed to verify and the counterexample option is enabled. Moreover, Silicon did produce a counterexample in the new representation. Before we can do any sort of processing, we need to know which functions of our Rust program failed. Therefore we need its so-called `DefId`, which uniquely identifies a function. To obtain the function's `DefId`, we make use of the existing error manager of Prusti [13]. Its purpose is to map errors in Silicon back to the element in Rust that caused it, to report the failing statement. For every generated Viper statement that might potentially cause a verification error, it registers an error location. They are registered during the encoding of the program, and we simply added the `DefId` of the currently encoded function to each of the registered errors. Whenever verification fails and Prusti identifies the reason for the failure, we are additionally given the identifier of the failing function.

4.2.2 Variables

After we find a failing function, we need to determine the variables that our counterexample should contain. We also want to know which of those variables are arguments of the function. Given the `DefId`, we can get the

MIR-encoding of the function that fails verification. The variable names of the generated Viper program are the ones that we find in the MIR, however, to give a counterexample we need to know which original Rust variables they correspond to. To find the variables of interest we use a data structure given by the compiler called `VarDebugInfo`, giving us a mapping from source Rust variables to their corresponding MIR variables, for example x -> _1, y -> _2. They are always numbered starting from 1 and we are also given the number of arguments `arg_count`, telling us that the first `arg_count` variables are the function's arguments. Additionally, we know that the variable _0 always stores the result of the function. Given the MIR identifier of a variable, we can also get its type which means now, given a Silicon model, we can try to get a value for this variable.

4.2.3 Backtranslation

In this section, we look at how to translate an extracted model entry back to a value of a Rust variable. For this part of the implementation, we need to be aware of the fact that a given model might not contain the entry for our variable at all or that its entry might not contain all the information we need, e.g. some fields of a reference might not be available. Our goal is to get as much information as possible from the given entries. The result of the translation is a new type of entries, implemented as an enum with

Listing 4.5: Definition of Prusti's counterexample entries

```rust
pub enum Entry {
    IntEntry { value: i64 },
    BoolEntry { value: bool },
    CharEntry { value: char },
    RefEntry { el: Box<Entry> },
    Struct {
        name: String,
        field_entries: Vec<(String, Entry)>,
    },
    Enum {
        super_name: String,
        name: Option<String>,
        field_entries: Vec<(String, Entry)>,
    },
    Tuple { fields: Vec<Entry> },
    Unit,
    UnknownEntry,
}
```

one variant for each of the supported types with the addition of unknown entries, which represent missing information or unsupported types. From now on we will always refer to Silicon's extracted model-entries as *Silicon entries*, and the ones we just introduced as *Rust entries*. The definition of the Rust entries is given in Listing 4.5.

Assume we are given an identifier of a local MIR variable, its Rust type and an extracted Silicon model, and we want to get a value for the counterexample of the variable. Since the names of the MIR variables are the same as their translated Viper variables, we search the model for an entry of our current variable name. If we find an entry we translate the found Silicon entry according to the given type-definition [2] of the Rust variable. Otherwise, we return an `UnknownEntry`. In the following sections, we discuss how the various Rust types are translated to Viper and how we can back-translate a Silicon entry to a Rust entry given the type it encodes. More details on the Rust-Viper encoding performed by Prusti can be found in the main publication [3] or the developer guide [12].

Primitive Types

The encoding of integers, booleans and characters is straightforward. Any type of integer (unsigned, signed, long, short) in Rust is usually encoded to a reference with a field `val_int` in Viper. However, the generated encoding is sometimes simplified to just an integer. Depending on the Silicon entry we find, we either directly use the contained value, or if we get a reference we look for the field `val_int` and get its value. If we do find the field we return an `IntEntry` containing the found value and otherwise an `UnknownEntry`. Two examples of given Silicon entries and the internal representation of the translated Rust entries are given in the following:

```
_1 <- 21
_2 <- Ref ($Ref!val!3) {
    val_int <- 43
}
```

Listing 4.6: Silicon

```
_1 <- IntEntry(21)
_2 <- IntEntry(43)
```

Listing 4.7: Prusti

The translation of booleans works analogously, with the sole difference that the field is named `val_bool` and we return a `BoolEntry`. For characters, there is a difference because Viper does not have a char type. Rust characters are translated to integers in Viper, meaning we first get an integer value from the counterexample, as before. This value then has to be converted back to a `char` and we return a `CharEntry`. The following listings show an example of the translation of chars and booleans. Note that the entry for `_1` looks

exactly the same as for an integer, but because the target variable in Rust is of type char it is translated differently:

```
_1 <- Ref {
    val_int <- 63
}
_2 <- Ref {
    val_bool <- true
}
```

Listing 4.8: Silicon

```
_1 <- CharEntry('c'),
_2 <- BoolEntry(true),
```

Listing 4.9: Prusti

References

Rust references are, not very surprisingly, encoded to Viper references. They have a field `val_ref` containing the value they point to. The provided type-definition also gives us the type of the value it points to. Given the Silicon entry generated for a Rust reference, we get its field `val_ref` and then recursively translate it according to the reference's sub-type. Then we return a `RefEntry` containing the result of the recursive translation. In the following listings, we show an example Silicon entry and the result of the translation for the type `Ref(i32)`:

```
_1 <- Ref {
    val_ref <- Ref {
        val_int <- 0
    }
}
```

Listing 4.10: Silicon

```
_1 <- RefEntry(IntEntry(0))
```

Listing 4.11: Prusti

Tuples

Tuples in Rust are also translated to references in Viper and their elements are stored in fields named `tuple_0`, `tuple_1`, etc. The type information gives us a list of the types of all the elements. Depending on the length of this list we try to find all the entries for those fields of the reference, and again translate the entries we find recursively. If we do not find a field, the difference to the previous types is that only the element that can not be found will be an `UnknownEntry`. Translating all the fields gives us a vector of Rust entries which we then return within a `TupleEntry`. In the example below, we show a Silicon entry that is generated for a tuple of the type `Tuple(i32, bool)`. Note that the references pointed to by `tuple_0` and `tuple_1` have the same

fields, which comes from the fact that they both point to the same reference.

```
_1 <- Ref {
  tuple_0 <- Ref {
    val_int <- 32
    val_bool <- false
  }
  tuple_1 <- Ref {
    val_int <- 32
    val_bool <- false
  }
}
```

Listing 4.12: Silicon

```
_1 <- TupleEntry([
    IntEntry(32),
    BoolEntry(false)
])
```

Listing 4.13: Prusti

Algebraic Data Types

Enums and structs in Rust belong to the same category called algebraic data types (ADT). Unions are also part of the same class but they are not supported by Prusti yet. We focus on structs first since they are just enums with only one variant in Rust. A struct is translated to a reference with a field for each field of the struct. The type definition provides us with the name of the struct and all the names of the fields and their types, which lets us find the Silicon entries for the fields and evaluate them recursively again. Listing 4.14 shows an example of a struct definition in Rust.

Listing 4.14: Example of a struct definition

```
struct SomeStruct {
    value: i32,
    other_value: i32,
    valid: bool,
}
```

The Silicon entry for an instance of the given struct has the structure of the following example and translating it gives us the shown Rust entry.

```
_0 <- Ref {
    f$valid <- Ref {
        val_bool <- true
        val_int <- 43
    }
    f$value <- Ref {
        val_int <- 42
    }
    f$other_value <- Ref {
        val_bool <- true
        val_int <- 43
    }
}
```

Listing 4.15: Silicon

```
_0 <- Struct {
  name: "SomeStruct"
  fields: [
    value: LitIntEntry(42),
    other_value: LitIntEntry(43),
    valid: BoolEntry(true)
  ]
}
```

Listing 4.16: Prusti

To extend this translation to enums, we will consider the following example definition:

```
enum Choose {
    One,
    Two{x: i32, y: bool},
    Three(char, bool)
}
```

To encode an enum, Prusti generates a reference with fields for each variant and an additional field storing the enum's discriminant. Each variant is encoded like a struct. The discriminant is an integer that has a certain value for each variant. An example of a Silicon entry that was generated for the above enum is shown in Listing 4.17.

To translate the Silicon entry of an enum, we always try to get its discriminant first because it determines the variant of the enum. For our current example, we get the value 2 for the discriminant. Given a discriminant and the internal representation of the enum definition, we can get the name of the variant and the names of all its fields. In this case, the discriminant tells us that the Silicon entry represents an instance of the variant `Three`. Therefore, we have to get the entry of the field `enum_Three` and evaluate it as if it was a struct. For the given example we will encounter recursive references, which means we have to look for the complete reference entry with the same identifier to resolve some of the fields. Finally, we get the following Rust entry:

Listing 4.17: Examples of a Silicon entry generated for enum `Choose`

```
_1 <- Ref ($Ref!val!0) {
    enum_Two <- Ref ($Ref!val!2) {
        f$0 <- RecRef($Ref!val!2)
        f$1 <- RecRef($Ref!val!2)
        val_int <- 99
        val_bool <- false
    }
    discriminant <- 2
    enum_Three <- Ref ($Ref!val!2) {
        f$0 <- RecRef($Ref!val!2)
        f$1 <- RecRef($Ref!val!2)
        val_int <- 99
        val_bool <- false
    }
}
```

Listing 4.18: translated Rust-entry

```
Enum {
    super_name: "Choose",
    name: "Three",
    field_entries: [
        0: CharEntry('c'),
        1: BoolEntry('false'),
    ]
}
```

4.2.4 Choosing Labels

At this point, we have a set of Rust variables for which we want values in the counterexample, their corresponding MIR and Viper variable names and the capability to translate Silicon entries back to Rust entries. We now have to choose the models of the Silicon counterexample from which we want to get the values for the entries. To get the values of all variables at the end of the program's execution, the intuitive choice is to use the main model and for the initial values of the arguments, the most obvious choice would be the model at the label "old".

However, after implementing this part of the project, we noticed that choosing the correct model is more subtle given the internal workings of Silicon. The problem is that in Viper, to access fields of references, one requires the permission to do so. In the programs generated by Prusti, most fields are accessed via predicates that are unfolded when those values are accessed

and folded again at a later point. At any label of the Viper program, if the predicate granting permission to the fields of a reference has not been previously unfolded or is already folded again, the counterexample does not know the field's values. To put this into context we consider the following Rust function that takes a struct (cf. Listing 4.14) as an argument:

```rust
pub fn foo(x: SomeStruct) {
    assert!(x.value == x.other_value || x.valid)
}
```

As we already know, the struct x will be translated to a reference _1 with multiple fields. We will now look at a very reduced version of the generated Viper method, containing only the statements that are relevant for whether or not fields of _1 are accessible, which is shown in Listing 4.19.

Listing 4.19: Predicate definition, labels and statements involving permissions of Viper program generated by Prusti for the Rust function in Section 4.2.4

```
1  predicate m_SomeStruct$_beg_$_end_(self: Ref) {
2    acc(self.f$value, write) && (acc(i32(self.f$value), write)
3      && (acc(self.f$other_value, write)
4      && (acc(i32(self.f$other_value), write)
5      && (acc(self.f$valid, write)
6      && acc(bool(self.f$valid), write)))))
7  }
8
9  method m_foo() returns (_0: Ref)
10 {
11   var _1: Ref
12   label start
13   inhale acc(m_SomeStruct$_beg_$_end_(_1), write)
14
15   label pre
16   inhale true
17   unfold acc(m_SomeStruct$_beg_$_end_(_1), write)
18   unfold acc(i32(_1.f$value), write)
19
20   label l0
21   unfold acc(i32(_1.f$other_value), write)
22
23   label l1
24   unfold acc(bool(_1.f$valid), write)
25
26   label l4
27   ...
28 }
```

m␣SomeStruct$␣beg␣$␣end␣ is the name of the predicate that gives access to the fields of the struct once it is unfolded. Since this unfolding (line 17) takes place after the label pre (line 15), these fields will only be in the counterexample for the models at the labels after it. The access to the fields of the actual integer and boolean values is unfolded later. Only at label l4 and after will all values of the struct be available. We can see this in the following entries of the generated Silicon counterexample at the various labels:

Listing 4.20: Silicon entries for ␣1 of Listing 4.19 at the various labels

```
// at label old, start and pre:
_1 <- Ref ($Ref!val!0) {
}
// at label l0:
_1 <- Ref ($Ref!val!0) {
    f$valid(perm: 1/1) <- Ref ($Ref!val!3) {
        val_int(perm: 1/1) <- 0
    }
    f$other_value(perm: 1/1) <- Ref ($Ref!val!2) {
    }
    f$value(perm: 1/1) <- Ref ($Ref!val!3) {
        val_int(perm: 1/1) <- 0
    }
}
//at label l4:
_1 <- Ref ($Ref!val!0) {
    f$valid(perm: 1/1) <- Ref ($Ref!val!3) {
        val_bool(perm: 1/1) <- false
        val_int(perm: 1/1) <- 0
    }
    f$other_value(perm: 1/1) <- Ref ($Ref!val!2) {
        val_int(perm: 1/1) <- 1
    }
    f$value(perm: 1/1) <- Ref ($Ref!val!3) {
        val_bool(perm: 1/1) <- false
        val_int(perm: 1/1) <- 0
    }
}
```

In a similar fashion, if we want to get a value for the result of a function we can usually not use the main model because at this point the permissions for the fields of the result are already folded again. Unfortunately, there is no simple rule for where to evaluate variables such that we can be certain that we get a complete entry. In some cases, the unfolding and folding even

happens within the same label, such that the values can not be obtained at any label. To solve this problem we considered multiple solutions:

- when verification fails, insert additional fold/unfold statements at specific places and rerun verification to get more information;

- use the control flow graph of the program to find the labels where the values should be unfolded and evaluate them there; or

- change Silicon's implementation in a way that makes these values accessible even when they are not unfolded;

We dismissed the first idea because it would clutter the translation and double the running time. The second idea would still not always work, because in some cases the values are not available at any label as previously mentioned. Even though we might get more information than we currently do, we decided against this approach because it is more of a workaround than an actual solution to the problem. The third idea could potentially make all the values available at any label. Unfortunately, the changes required in Silicon are quite involved and were not feasible in time for our project but they might be part of another project in the future.

For now, Prusti simply evaluates its arguments at the label `l0` and all variables (including the arguments) at the last label that can be found in the list of models. Evaluating the arguments at the label `l0` makes sense because at that point mutable arguments are usually not modified yet and we can often get some useful information already. We choose the last labeled model instead of the main model because the main model almost never contains the value for the result even though we might not get the true final value of a variable in some cases. The final result, after translating the given models as we have just described, gives us two Rust entries for each argument and one entry for all other local variable and the result.

4.2.5 Presentation

The last section of this chapter describes how we present the counterexample to the user. Note that all the results of the back-translation we have previously seen explicitly showed the internal representation of Rust entries and not how they are presented to the user. Their actual output is close to how their values would be written in Rust. For example, the Rust entry that was the result of the translation of a struct in Listing 4.16 is presented in the following way:

```
x <- SomeStruct {
    value: 0,
    other_value: 1,
    valid: false,
}
```

We will show more examples of different types during the evaluation in Chapter 5. As mentioned in Chapter 3, the entries are presented as compiler notes. For each variable, we print a note containing the entries and it's span. The compiler also automatically outputs the line of code where the variable is defined and underlines the variable. While the initial reason to use compiler notes was to make the output compatible with Prusti-Assistant, it has the additional advantage that we can easily identify variables in case of duplicate names. For example, the function in Listing 4.22 generates the following two compiler notes for the two variables named x:

Listing 4.21: Prusti's presentation of counterexamples

```
note: counterexample for "x"
x <- 7
  --> counterexample-thesis-resources/cetests/ref2.rs:10:9
   |
10 |          x => x * 2
   |          ^
note: counterexample for "x"
initial: x <- ref(7)
final: x <- ref(7)
  --> counterexample-thesis-resources/cetests/ref2.rs:5:8
   |
5  | fn foo(x: &mut i32) -> i32{
   |        ^
```

From this output it is clear which variable each note corresponds to. If we wanted to present the counterexample in a similar fashion to Nagini we would have to interpret the spans of variables in case of duplicate names. While the generated notes are currently not used by the VSCode extension Prusti-Assistant [4], it still recognizes and displays them in a more compact way as shown in Figure 4.2. Clicking a note makes the cursor jump directly to the variable it refers to. In the future this should eventually allow for a better IDE integration, where the variables would actually be underlined in the source code and hovering over them would display the generated counterexample.

```
#[ensures(result != 14)]
fn foo(x: &i32) -> i32{
    let y = *x;
    match y {
        x => x * 2
    }
}

fn main(){}
```

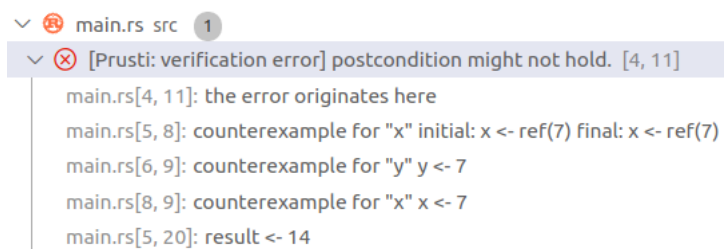**Listing 4.22:** Rust example with duplicate names

Figure 4.2: Generated list of problems in VSCode when using Prusti-Assistant with our Version of Prusti and counterexamples enabled

Chapter 5

Evaluation

The evaluation of our counterexample implementation in Prusti consists of two parts: a qualitative evaluation of the generated counterexamples and an analysis of their influence on Prusti's performance.

5.1 Qualitative Evaluation of Counterexamples

We tested our implementation on a handpicked collection of Rust programs that fail verification in Prusti. Initially, our plan was to evaluate the quality of the counterexamples using Prusti's existing tests, namely those that fail verification. Before starting the implementation, we annotated all of those tests with what we would consider optimal counterexamples. However, due to the limitations of our implementation caused by the often inaccessible values of counterexamples (cf. Section 4.2.4), we chose examples that illustrate the strengths and weaknesses of our implementation. We evaluate

Example	Complete	Spurious	RE	LOC	VT	Source
sum.rs	✓	✗	✓	10	0.49	Prusti tests [13]
replace.rs	✓	✗	✓	13	0.46	
tuple.rs	✗	✗	✓	8	0.71	
account.rs	✗	✗	✓	18	1.05	
enum.rs	✗	✗	✓	13	0.80	
account-fail.rs	✗	✗	✗	11	0.51	
loop.rs	✓	✓	✗	11	0.75	

Table 5.1: The list of examples used for the evaluation and a summary of the attributes of their generated counterexamples. We evaluated their completeness, i.e. the counterexample contains no missing values, and whether the counterexample lets us reproduce the error (RE). It also contains the example's lines of code (LOC) and verification time (VT) in seconds with counterexamples enabled.

the back-translation for various types, discuss the validity of the generated counterexamples and then showcase the limitations of our implementation. Table 5.1 summarizes the results. Let us consider each example in more detail.

sum.rs Going back to our very first example in the introduction, shown again in Listing 5.1, we can see the generated output of the new Prusti version in Listing 5.2. This function is part of Prusti's test suite and also shows our manual annotation of an "ideal counterexample" as we previously discussed. The generated counterexample is valid since it gives us a negative value for n and the program returns 0 for that value, which indeed violates the post-condition. Therefore this example does meet our initial expectations. The generated output is comparable to other compiler errors (e.g. invalid borrows) and thus the format is familiar to Rust programmers. Nonetheless, a more concise error message in an IDE (e.g. as we have seen in Figure 2.1 for Dafny) might be preferable. For the rest of the evaluation, we will not be showing all of the generated output and reduce it to only the information about the variables.

Listing 5.1: sum.rs (function from Prusti's test suite with counterexample annotation)

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

Listing 5.2: Prusti's output for sum.rs

```
error: [Prusti: verification error] postcondition might not hold.
 --> counterexample-thesis-resources/evaluation/sum.rs:5:11
  |
5 | #[ensures(result == n*(n+1)/2)]
  |           ^^^^^^^^^^^^^^^^^^^^^
note: counterexample for "n"
initial: n <- -2
final: n <- -2
 --> counterexample-thesis-resources/evaluation/sum.rs:6:8
  |
```

```
6 | fn sum(n: i32) -> i32 {
  |            ^
note: result <- 0
 --> counterexample-thesis-resources/evaluation/sum.rs:6:19
  |
6 | fn sum(n: i32) -> i32 {
  |                         ^^^

Verification failed
```

replace.rs Next we consider an example involving references, chars and
booleans. Listing 5.3 contains a function that checks if the given reference
points to the char '$'. If it does, depending on whether the argument
acc is true, it mutates the char to a white-space or panics. The generated
counterexample, given in Listing 5.4, provides us with the correct arguments
leading to a panic. It also illustrates how we decided to present chars. Apart
from the char itself, we also display its hexadecimal value. Additionally, we
note that the counterexample gives us a value for the result even though the
function would encounter a panic, an unrecoverable error in Rust, before
returning any result. Our implementation currently does not detect whether
the reason for the verification error is part of the function (e.g. assertion
error, panic, etc.) or its post-condition and tries to get a value for the result
in both cases, even when it should not be part of a counterexample.

Listing 5.3: replace.rs

```rust
fn replace(x: &mut char, acc: bool) {
    match x {
        '$' => {
            if acc {
                *x = ' ';
            } else {
                panic!("no␣access");
            }
        },
        _ => {}
    }
}
```

Listing 5.4: Generated counterexample for replace.rs

```
initial:
    acc <- true
    x <- ref('%' (0x25))

final:
    acc <- true
    x <- ref('%' (0x25))
    result <- ()
```

tuple   The next function, shown in Listing 5.5, is given a tuple as an argument with the precondition that the first element of the tuple has to be larger than zero and the contained char has to be 'c'. It returns another tuple where the two entries of the argument are swapped and two is subtracted from the integer field. We try to prove that the second field of the result is always greater than or equal to zero.

The resulting counterexample is shown in Listing 5.6. Looking at the initial value of the argument we can see that the first field of the tuple is available and correct since 1 is the only value that does not violate the precondition but leads to a violation of the postcondition. The question mark in the

Listing 5.5: tuple.rs

```rust
#[requires(x.0 > 0 && x.1 == 'c')]
#[ensures(result.1 >= 0)]
fn foo(x: (i32, char)) -> (char, i32) {
    let y = x.0 - 2;
    let z = x.1;
    (z, y)
}
```

Listing 5.6: Generated counterexample for tuple.rs

```
initial:
    x <- (1, ?)

final:
x <- (1, 'c' (0x63))
y <- -1
z <- 'c' (63)
result <- ('c' (0x63), -1)
```

second field represents an `UnknownEntry` indicating that the corresponding Viper field has not been unfolded at label `10` where the initial values are evaluated. However, in the set of final values of the counterexample, we can still find its value which is also correct. The result and the intermediate values also match an execution of the function and therefore this is also a valid counterexample.

account.rs   For structs, we consider the function in Listing 5.7. The function `transfer` takes in two accounts and transfers an amount from account `x` to account `y`. The post-condition it tries to prove does not hold and finding a counterexample is trivial. However, this time we are handling mutable arguments and are interested in the values for `x` and `y` before and after they are mutated.

In the generated counterexample, given in Listing 5.8, we can see that most values are available. The initial `amount` and the initial balance of `y` is missing. The missing amount does not really pose a problem, since it is not mutated and must be equal to its initial value. We can see that the balance of `x` got reduced by the correct amount, but for `y` we could not be sure if we hadn't created the additional variable `temp` storing its value. Missing values are a major problem of our implementation, but in many cases, we can still deduct what the reason for a failure was if we analyze the available values.

Listing 5.7: account.rs

```
pub struct Account {
    balance: i32,
}

#[requires(amount > 0 && x.balance > amount && y.balance >= 0)]
#[ensures(old(y.balance) > result.1.balance)]
pub fn transfer(
    mut x: Account,
    mut y: Account,
    amount: i32
) -> (Account, Account) {
    if x.balance >= amount {
        let temp = y.balance;
        x.balance -= amount;
        y.balance += amount;
    }
    (x, y)
}
```

Listing 5.8: Generated counterexample for account.rs

```
initial:
    x <- Account { balance: 2701 }
    y <- Account { balance: ? }
    amount <- ?
    amount <- 2700

final:
    final: x <- Account { balance: 1 }
    y <- Account { balance: 5399 }
    temp <- 2699
    result <- (
        Account { balance: 1 }
        Account { balance: 5399 }
    )
```

enum.rs Moving on to the last supported type, Listing 5.9 shows a function that evaluates enums representing three binary operations. The operation that is causing the error is `BinOp::Div` because it can lead to a division by zero. The generated counterexample, shown in Listing 5.10 does give us this value for the argument, but once again the fields are not available in the initial entry. The counterexample also contains all the variables for each arm of the match statement. This is unpleasant since many of those variables are never created during an execution which makes the counterexample wrong and less readable. Resolving this would require some analysis of the paths taken but for any analysis of this sort, the counterexample for the arguments would have to be complete.

Listing 5.9: enum.rs

```
enum BinOp {
    Add(i32, i32),
    Sub(i32, i32),
    Div(i32, i32),
}
fn apply(op: BinOp) -> i32 {
    match op {
        BinOp::Add(a, b) => a + b,
        BinOp::Sub(c, d) => c - d,
        BinOp::Div(e, f) => e / f
    }
}
```

Listing 5.10: Generated counterexample for Listing 5.9

```
initial:
    op <- BinOp::Div(?, ?)
final:
    op <- BinOp::Div(2, 0)
    a <- 0
    b <- 0
    c <- 0
    e <- 2
    f <- 0
    result <- 0
```

Listing 5.11: account-fail.rs

```rust
fn get_balance(acc: Account) -> i32 {
    acc.balance
}


#[ensures(result)]
fn has_money(acc: Account) -> bool {
    get_balance(acc) > 0
}
```

account-fail.rs So far, even though we have mentioned the limitations of
our approach several times, we have only shown examples where the inac-
cessible values of a counterexample were not actually relevant to trace the
verification failure back to what causes it. However, in some cases we are not
able to derive the reason for the verification failure from the counterexample
and an example function for this behavior is given in Listing 5.11. The func-
tion has_money is the one failing verification. It operates on the same struct
Account that was defined in Listing 5.7. The post-condition is violated for
any account with a negative balance, but the important difference for this
example is that the balance is accessed via an additional function. Because

Listing 5.12: Generated counterexample for Listing 5.11

```
initial:
    acc <- Account { balance: ? }
final:
    acc <- Account { balance: ? }
    result <- false
```

of that, the fields of the Viper variable _1, the translated variable of `acc`, will never be unfolded in Viper and the value causing the failure is not accessible at any label of Silicon's counterexample. Therefore the generated counterexample, shown in Listing 5.12, does not contain any valuable information about the arguments.

loop.rs   Finally we discuss an example of a spurious counterexample. As mentioned in Chapter 2 spurious behavior is possible in cases, where the space of reachable abstract states contains an error state but there is no program execution, i.e. sequence of concrete states, realizing that error. Listing 5.13 shows an example where we expect this behavior. The function consists of a loop and a post-condition that will not be violated when executing the function. However, the loop invariant is too weak to prove that the assertion holds; therefore verification fails. After the loop, the verifier knows that the assertion `y > 0 && x >= 0 && !(x > 0)` must hold, which is an over-approximation of the concretely reachable states because the only reachable state is `x == 0 && y == 1`. Because of this over-approximation, there exists a solution to this equation that also violates the post-condition leading to a spurious counterexample. Prusti's generated counterexample gives us a final value of 16 for `y` which aligns with the expected behavior.

Listing 5.13: loop.rs

```rust
#[ensures(result < 16)]
fn spurious() -> i32 {
    let mut x = 10;
    let mut y = 1;
    while(x > 0) {
        body_invariant!(x >= 0 && y > 0);
        x = x - 1;
        y = y + 1;
    }
    y
}
```

5.2 Timing Analysis

For the evaluation of the influence on Prusti's performance when generating counterexamples, we adjusted Prusti's existing benchmarking script to run a set of 121 Rust files that fail verification with counterexamples enabled and disabled. The tests consist of the examples used during the evaluation and Prusti's existing tests that fail verification.

The results of this analysis showed that verification with counterexamples enabled takes on average around 0.1 seconds longer than when they are disabled. The distribution of the increase of the execution time in seconds is shown in Figure 5.1. In most cases, the difference is less than 0.2 seconds but there are a few outliers where the difference is quite substantial. Analyzing the extreme cases where the increase is more than 2 seconds showed that both of them involved loops, however, other examples involving loops did not confirm a trend of high verification times for loops in general.

Additionally, we were interested in how this increase is related to the total verification time. The results show that enabling counterexamples increases the verification time by on average 6.5%. Figure 5.2 shows that the influence can be quite substantial in some cases, but is smaller than 20% in most (>90%) of the tests.

The influence on the performance should be barely noticeable in most cases and should not have an influence on the verification experience. Since counterexamples will mostly be used as a debugging tool and not be constantly enabled, this seems acceptable.
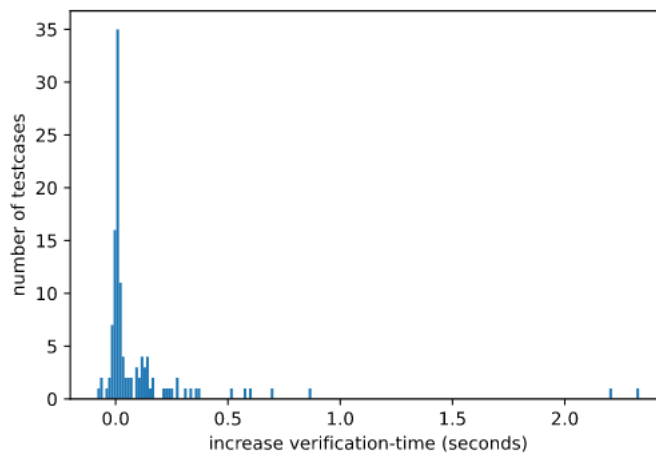

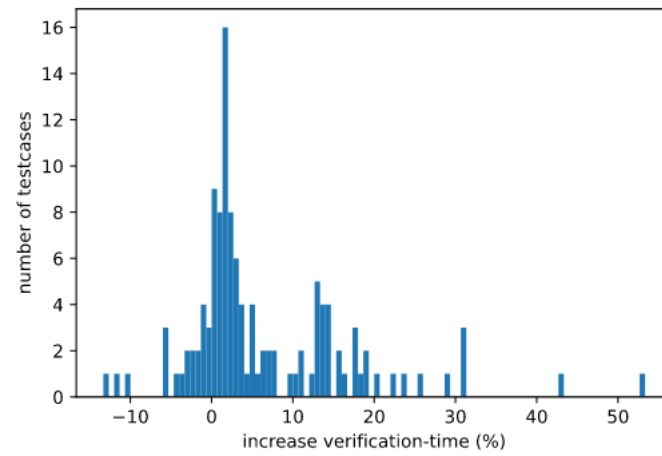**Figure 5.1:** Histogram for increase in verification time when using counterexamples

Figure 5.2: Histogram for relative increase in verification time

Chapter 6

Conclusion

The goal of this thesis was to produce counterexamples for Rust programs that fail verification. To this end, we improved the representation of counterexamples computed by Viper's Silicon back-end. Based on this representation we then implemented a back-translation to Rust variables, which yields the actual counterexample.

On the Silicon level, our implementation has already proven to be useful by simplifying the interaction with Prusti significantly. Comparing our implementation to Nagini's implementation using the native counterexamples shows that our representation shifts a lot of effort to the back-end. This claim is further supported by the fact that there are already exists an ongoing project using our implementation with the goal of extending Gobra [11], another Viper based verifier, to generate counterexamples.

We qualitatively evaluated our implementation of Rust counterexamples. They are already useful in some cases but have room for improvement. Our counterexamples are limited by the fact that relevant values are often missing in the generated Silicon counterexamples. However, as explained in Section 4.2.4, we think that this problem could be resolved on the Silicon level in the future. If the accessibility issue was resolved, we would expect our current implementation to produce meaningful counterexamples consistently, since our back-translation works well in the cases where the necessary information is available. The generated counterexamples are currently provided to the user via compiler notes, which is a familiar format for Rust programmers.

6.1 Future Work

There are several directions for future work that would either improve our current implementation of counterexamples or use them for some interesting analysis of the function under verification. Some of them are listed in the following:

- Extend Prusti-Assistant [4] to process the generated counterexamples and display them within the code in a similar fashion to what we have seen for Dafny [7].

- Extend the set of supported types, for example, boxes which would allow us to handle unbounded data structures.

- Extend the counterexamples to provide more fine-grained debugging information. For example, if we had the values of the variables at all labels, we could process them at each label and analyze the changes for each variable. Instead of getting just one value, we could get all the values of a variable during an execution (assuming they are not mutated twice in the same basic block).

- Generating counterexamples for the results of function calls would be another useful extension. At the moment, if the result of a function call is not stored in a local variable, this value will never be a part of the counterexample. If a function is non-deterministic and the verification error depends on its return value, we can not reproduce the failure. Extending our current implementation with this functionality should be a relatively easy adaption since the MIR encoding will create a temporary variable storing the result of each function call, and therefore the same is true for the translated Viper encoding.

- Given a counterexample, one could automatically run the function under verification for the provided arguments. A suitable tool to implement this functionality is Miri [9], a MIR interpreter. Executing a program for the given counterexample could be used to check whether a counterexample is realizable or caused by a too coarse specification.

Bibliography

[1] The rust programming language. https://www.rust-lang.org. Accessed: 28.04.2021.

[2] Rust type definitions. https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/ty/sty/enum.TyKind.html. Accessed: 02.05.2021.

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

[4] Julian Dunskus. Developing ide support for a rust verifier. Master's thesis, ETH Zürich, 2020.

[5] Marco Eilers and contributors. Repository containing nagini's source code and documentation. https://github.com/marcoeilers/nagini. Accessed: 05.05.2021.

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

[7] Luke Herbert, K. Rustan M. Leino, and Jose Quaresma. Using dafny, an automatic program verifier. In *LASER Summer School*, volume 7682 of *Lecture Notes in Computer Science*, pages 156–181. Springer, 2011.

[8] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Dependable Software Systems Engineering*, volume 50 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 104–125. IOS Press, 2017.

[9] Scott Olson and Christopher Dutchyn. Miri: An interpreter for rust's mid-level intermediate representation. 2016.

[10] Cedric Stoll. Smt models for verification debugging. Master's thesis, ETH Zürich, 2019.

[11] Gobra team. Repository containing gobra's source code and documentation. https://github.com/viperproject/gobra. Accessed: 03.05.2021.

[12] Prusti team. Prusti developer guide. https://viperproject.github.io/prusti-dev/dev-guide. Accessed: 02.05.2021.

[13] Prusti team. Repository containing prusti's source code and documentation. https://github.com/viperproject/prusti-dev/. Accessed: 28.04.2021.

[14] Viper team. Repository containing silicon's source code. https://github.com/viperproject/silicon. Accessed: 05.05.2021.

[15] Viper team. Viper tutorial. http://viper.ethz.ch/tutorial/. Accessed: 04.05.2021.

ETH

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 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):**
Hegglin	Cedric

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)**
Zürich, 05.05.2021	C. Hegglin

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