Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Advanced Counterexample Generation in Viper

Bachelor Thesis

Raoul van Doren

July 8, 2023

Advisors: Prof. Dr. Peter Müller, Dr. Marco Eilers, Aurel Bílý

Department of Computer Science, ETH Zürich

Abstract

Automated program verification is an essential feature for the reliability of code correctness. The Viper intermediate verification language offers a verifier for permission-based reasoning. Still, its error identification mechanism does not provide an exhaustive method for generating counterexamples if verification fails. This thesis addresses this limitation by improving Viper's counterexample generation of both Viper backends and enhancing its debugging procedure.

Two main objectives drive this project. The first is to establish a common counterexample representation with all required features independent of the backend. The second goal involves augmenting the comprehensiveness of counterexamples by including various types currently absent from the Viper language, such as fields, predicates, wands, and quantified permissions.

The anticipated outcome is a significant enhancement of Viper's utility as a program verification tool, offering developers a more intuitive and efficient debugging process.

Acknowledgements

I want to thank my supervisors Dr. Marco Eilers and Aurel Bílý for providing valuable feedback during our weekly meetings and consistently supporting me. Additionally, I want to thank Prof. Dr. Peter Müller for granting me the opportunity to work on this exciting project and be a part of Viper's research group.

Contents

Chapter 1

Introduction

Automated program verification has become essential to ensure code correctness and compliance with specified behaviors. Among the tools available for this process, Viper [11] has established itself as a robust verifier for permission-based reasoning. Developed by the Programming Methodology Group at the Department of Computer Science at ETH Zurich, Viper is an intermediate verification language that forms the foundation of automatic verifiers for several programming languages, including Python (via Nagini [4]), Rust (via Prusti [18]) and others. Viper uses two backends to verify the correctness of programs: the Symbolic Execution backend [13], internally called Silicon, and the Verification Condition Generation backend [6], internally called Carbon.

These tools can verify the validity of programs' pre- and post-conditions, assertions, and permission properties. When verification fails due to errors in the program or annotations, it is often difficult to find the cause of the error, especially in complex programs. This presents an obstacle for the developer, who must often find the source of the error through a process of deduction and trial and error.

Counterexamples – sets of inputs that result in verification errors – can significantly accelerate the process of finding the error. By clearly pointing out the problem, counterexamples provide insight into the conditions under which the program behaves unexpectedly. This feature provides immediate feedback to the developer and enables more efficient problem identification and resolution.

Despite its potential, the current implementation of Viper lacks a comprehensive approach to generating counterexamples when verification fails. This results in developers receiving only limited information in case of verification failures. This work addresses this gap by improving the counterexample generation mechanism for Viper. The approach introduces a common

format for representing counterexamples, ensuring compatibility with both the Silicon and Carbon backends. However, it is important to note that the extraction mechanisms for generating these counterexamples are mostly separate for each backend. By presenting explicit counterexamples in a common format, this feature promises to extend the functionality of Viper and provide a more intuitive debugging process for developers.

This work stems from the existence of a relatively comprehensive counterexample format that is specific to the Silicon backend [14] [5] [1] of Viper. However, this format lacks some critical features. As a result of this previous work, there is an extraction mechanism for Silicon, but only a very basic one for the Carbon backend. The purpose of this work is, therefore, to bridge these gaps.

The primary goal is to define a common, backend-independent counterexample representation that can incorporate all important features. This would require extending the extraction process for Silicon, and building up an extraction process for the Carbon backend from scratch.

The secondary goal, once this universal counterexample representation is established, is to make the counterexamples more comprehensive. To this end, we will extend counterexample support to types in the Viper language that are not currently included. Specifically, we aim to include fields, predicates, wands and quantified permissions in the counterexamples, features that should greatly enhance the utility of Viper.

The methodological approach in this work is to identify the counterexample values in the verification feedback received from the SMT solvers, implement the necessary features to compactly gather the information for the counterexample, and represent them in a backend-independent counterexample format.

By achieving these goals, this work should significantly increase the usefulness of Viper as a program verification tool and provide developers with a more intuitive and efficient debugging process.

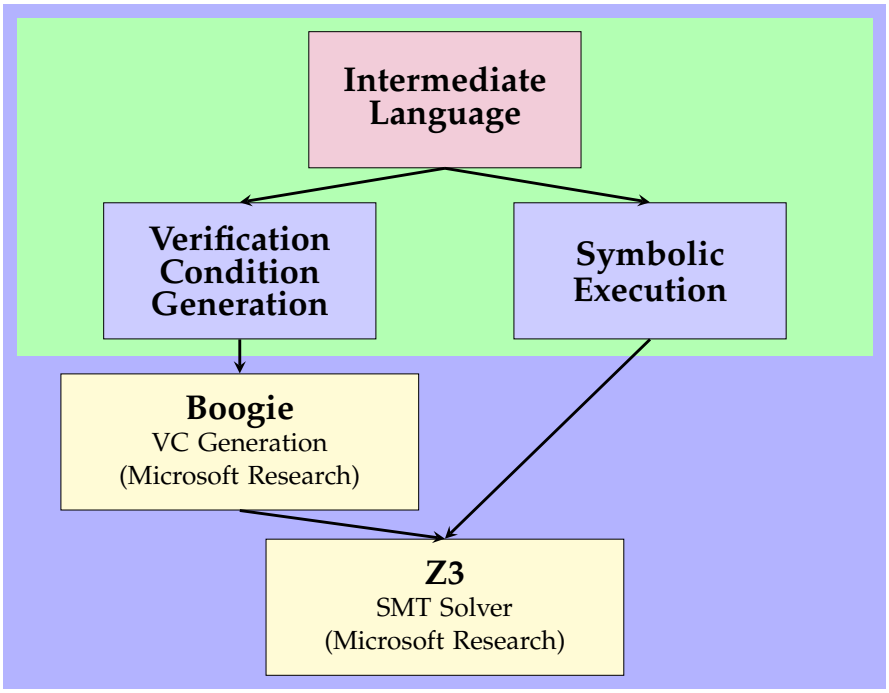
The next chapter provides the background on Viper, Silicon, Carbon, and counterexamples. Chapter 3 presents the new common counterexample format. Chapter 4 explains the added features to improve Viper's functionality. Chapter 5 discusses the information extraction processes for these features. In Chapter 6, we evaluate the accuracy and the overhead of the new system. Finally, Chapter 7 summarizes the project and suggests areas for future work.

Chapter 2

Background

In this chapter, we will discuss the technical background of this thesis. Firstly, we will introduce the Viper intermediate verification language along with its two backends: Carbon and Silicon. When verifying a program in Viper, we have the flexibility to select which backend, Carbon or Silicon, should be used. Furthermore, we will explain the features previously supported for counterexamples generated in Viper [1].

Figure 1: Viper infrastructure with its two backends and their externally used verification programs.



[1] release 23.07

2.1 Viper

Viper, a verification infrastructure for permission-based reasoning, is a programming language and toolkit developed at ETH Zurich for program verification. As a sequential, imperative intermediate verification language, it is used to verify partial and total correctness of program statements. In other words, it provides verification guarantees that the properties specified hold in a particular program state when that state is reached.

Viper supports mathematical types, user-defined predicates, functions, and several other unique features. In addition, Viper is designed to validate pre- and post-conditions and assertions relying on permission-based reasoning. Viper supports basic data types [15] such as integers (`Int`), Booleans (`Bool`), sequences (`Seq`), sets (`Set`), multisets (`Multiset`) and maps (`Map`) and additionally allows users to define their own types using domains. A program in this language contains global declarations for methods, fields, predicates, functions and user-defined domains.

We will now explore a selection of the Viper features as they can occur in a counterexample for specific verification failures. Note that the features discussed here are a subset of Viper's full capabilities, chosen for their relevance to the updated counterexample generation.

Permissions in Viper control access to the program heap, allowing for simpler framing (proving that an assertion is not affected by heap modification) and reasoning about concurrency. This means permissions help manage access to certain locations within a program during a method execution or a loop iteration.

Predicates [11] [12] in Viper consist of a name, a list of parameters, and a body. This body contains the assertion defining the predicate and can be optional, resulting in an abstract predicate that hides implementation details. The manipulation of predicates is handled via unfold and fold statements.

Viper, in addition to standard first-order logic and separation logic operations, utilizes a binary connective known as a Magic Wand [16]. Magic wands denotes guarantees about future additions to the state. They can be exchanged for another assertion when merged with a state that satisfies one assertion. They are employed in Viper to represent permissions to partial data structures.

Quantified permissions [11] are supported in Viper for specifying unbounded heap structures. These allow pointwise specification of permissions. They are especially useful for specifying data structures that are not strictly hierarchical, such as cyclic lists, arrays, and general graphs.

A Viper function [2] [10] consists of a name, parameters, preconditions, post-conditions, and a function body that is a single expression. Functions in

Viper can contain recursion and there cannot be any modifications to the program state.

Viper domains [2] [10] serve as a flexible tool for introducing new types and mathematical functions on these types. A domain consists of a name, domain functions and axioms. Unlike conventional Viper functions, domain functions lack preconditions, postconditions, or a body, making them total functions. The output of domain function evaluations is established through axioms.

Viper is intended to enable the development of new verification tools for common programming languages. This is accomplished through the encoding of the semantics and specifications of front-end programming languages into the Viper intermediate language, which enables the development of tools such as Gobra [17], Prusti [18], and Nagini [4] for Go, Rust, and Python, respectively.

2.2 Silicon

We will now look at Silicon, one of the two backends of the intermediate verification language Viper. Silicon is an automatic verifier for the Viper language that uses symbolic execution-based reasoning.

The symbolic execution approach allows Silicon to examine the possible states of a program by using symbolic variables that represent possible values. This way, multiple execution paths can be analyzed simultaneously.

To perform symbolic execution, Silicon maintains a symbolic store and a heap. The symbolic store is a mapping of each local variable to a symbolic value. This symbolic value represents embodies a collection of constraints that the variable can satisfy. The heap, on the other hand, contains heap chunks that represent the possible states of heap types in the program.

As Silicon symbolically executes the program, it traverses different paths in the program. While traversing each path, it collects a series of conditions known as path conditions. These conditions include all the facts that are known to be true on the particular execution path.

Upon reaching an assertion in the program, Silicon uses these collected path conditions to verify the assertion. It checks whether the known path conditions logically imply that the assertion is true. This means that if all the conditions collected along the path hold, the assertion should also hold true.

Silicon uses the Z3 SMT (satisfiability modulo theories) solver [3] to evaluate the veracity of program assertions through logical queries resulting from the symbolic execution of the program. It either confirms their validity or

points out possible errors in the program logic, based on the path conditions collected and the assertions made in the program.

2.3 Carbon

As mentioned earlier, Carbon is one of the two available backends for the Viper verification language. Carbon is a verification-condition-generation-based verifier for Viper that provides a means for program verification. Same as its counterpart Silicon, Carbon is based on modular verification and works in several steps.

First, it takes a program written in the Viper language as input. Then, it translates this program into a Boogie [8] file, which serves as an intermediate form used for the verification process. Boogie encodes the Boogie program into SMT form and passes it to the SMT solver. Finally, the SMT solver evaluates this newly encoded form to verify the assertions' truth against the specified conditions.

2.4 Counterexamples

When a verification error occurs inside of a Viper program, it is because the SMT solver finds a formula satisfiable when it is asked to prove the negation of an assertion. That is, for an assertion "assert X", the SMT solver is tasked with proving "not X". If it succeeds, it implies that the assertion does not hold, and a verification error is reported. In such a case, the SMT solver provides a model that represents a counterexample at the SMT level. This model can be traced back to the original Viper code where the error occurred, serving as a counterexample to the failed verification attempt. Consequently, if the SMT solver returns "unknown", it signifies a successful assertion, where the SMT solver could not prove the negation of the assertion.

In the case of verification with Carbon and Silicon, the Z3 SMT solver executes the verification of each method separately, which allows the generation of a distinct counterexample for each error. The counterexample is comprised of three key objects: a list of store variables, a list of heap entries (including fields, predicates, magic wands and quantified permissions), and a list of all the function models.

With partial models, the SMT solver determines values for a subset of the variables in the formula that make the entire formula true. However, it might not assign values to all variables or function calls in the formula. Instead, for the ones that are not explicitly assigned, Z3 returns "unknown".

For more detailed information about how SMT solvers generate counterexamples at a lower level, please refer to Cédric Stoll's thesis [14].

Chapter 3

Common Counterexample View

This chapter examines the original and the new counterexample format. For the new counterexample format we will show how the program's counterexample information is represented in an intermediate counterexample representation and then in an extended counterexample representation, independent of the backends used for program verification. As noted in Section 2.3, the previous work on counterexamples generation for Silicon is much more advanced than for Carbon. Thus, besides updating Carbon counterexamples to support the same functionalities as Silicon's counterexamples, finding a common counterexample representation for both backends is essential.

3.1 Previous Counterexample Format

Firstly, let us examine the Viper program shown in Listing 2.1. The program takes a reference and a sequence of integers as input parameters and assigns the integer from the first index of the sequence to the field accessed through the reference from the input parameters. The verification fails because the assertion states that the field accessed through the reference "`r`" has to be equal to "`5`". While a programmer might find it relatively easy to determine the reason for the verification error from this program, this might not be the case for verification errors occurring in more complex programs.

Now let us look at how Carbon and Silicon originally handled the counterexample generation of this program. When running the Viper program with Carbon as the backend, it should be noted that prior to our project, Carbon's most advanced counterexample generation only reported values for current variables, as shown in Listing 2.2. When run with the original Silicon backend, the most advanced counterexample generation reported the values for the variable, the sequence contents, and the specific field access, including its permission status, as shown in Listing 2.3.

Identifiers like "T@U!val!8" and "$Ref!val!0" might occur in the counterexamples, as can be seen in Listing 2.2 and 2.3. These are internal names reported by Z3, the SMT solver used in Viper, and they denote specific values or states in the SMT solving process.

Listing 2.1: Viper program containing a field, a reference, and a sequence failing verification due to an assertion.

```
field f: Int

method try (r: Ref, s: Seq[Int])
  requires |s| > 2
  requires acc(r.f)
{
  var x: Int
  x := s[0]
  r.f := x
  assert r.f == 5
}
```

Listing 2.2: Carbon's counterexample generated for the program shown in Listing 2.1.

```
FailureContextImpl(Some(r -> T@U!val!7
s -> T@U!val!8
x -> 13))
```

Listing 2.3: Silicon's counterexample generated for the program shown in Listing 2.1.

```
    counterexample:
model at label old:
r <- Ref ($Ref!val!0) {
  f(perm: 1/1) <- 0
}
s <- (Seq<Int>!val!1): [1, 1, 1]
x <- 1
on return:
r <- Ref ($Ref!val!0) {
  f(perm: 1/1) <- 1
}
s <- (Seq<Int>!val!1): [1, 1, 1]
x <- 1
```

Some types and heap extraction for counterexamples in Silicon have already been implemented before our work thanks to Cedric Hegglin's work, detailed in "Counterexamples for a Rust Verifier" [5]. His thesis centered around extending Prusti, a verification tool for Rust program, by integrating counterexample support into the verification. To do so, he mainly concentrated on developing Silicon's counterexample support for fields.

Fabio Aliberti's work, described in his thesis titled "Counterexample Generation in Gobra" [1], mainly focused on incorporating counterexample support, a verification tool for Go programs, but also played a crucial role in adding counterexample support for functions and domains into Silicon.

3.2 New Counterexample Format

We have included two new counterexample formats: the intermediate and extended counterexample formats. The main difference between these two formats lies in how variables and values of the counterexample are presented. The decision to have two different versions was made to address the specific needs of the programmer.

As opposed to the intermediate counterexample format, each variable from the store or the heap is assigned to its corresponding AST nodes in the extended format. The extracted counterexample heap entries are also separated into their actual types, such as field, predicate, or magic wand.

Additionally, the way internal names are presented differs significantly between the two formats. In the intermediate counterexample, names of fields, variables, functions, etc., are already translated into their corresponding names from the Viper program. However, certain values, such as references with internal names, remain untranslated. For values with internal identifiers, the extended counterexample represents these using the name of the variable that was given to them by the programmer. This translation provides a more human-readable counterexample.

Nevertheless, it is crucial to note that the ability to translate an internal value into the programmers' variable name depends on the counterexample model received from the SMT solver. As explained in the example for the previous chapter, in some cases, certain internal identifiers cannot be translated because they are not defined or might be assigned to different names in the model provided by the SMT solver.

The intermediate and extended counterexample formats allow us to balance comprehensiveness and readability. The extended format helps developers to comprehend the counterexample easily. In contrast, the intermediate format provides more detailed information, making it valuable for in-depth analysis and understanding of the verification results.

To understand the accomplishments of the two new counterexample formats and primarily point out the differences between the intermediate and the extended counterexample generation, we look at a more complex program variation of the previous example from section 2.3. The new example program shown in Listing 3.1 defines the predicate "StructA" holding the permission for references to the fields "`this.x`" and "`this.y`". The method

"compare" unfolds the predicate and assigns the value of the first entry from sequence "s" to the field reference "this.x". After this, we assert that the references to both fields are equivalent, which leads to a verification error.

Listing 3.1: Viper program containing predicates and fields failing verification.

```
field x: Int
field y: Int

predicate StructA(this: Ref) {
  acc(this.x) && acc(this.y)
}

method compare (this: Ref, s: Seq[Int])
  requires |s| > 2
  requires StructA(this)
{
  unfold StructA(this)
  this.x := s[0]
  assert this.x == this.y
}
```

3.2.1 Intermediate Counterexample

The intermediate counterexample is generated in the IntermediateCounterexampleModel class for both backends. The counterexample is shown to the user when the verification of a program is called with the "--counterexample intermediate" flag.

We decided on adding an intermediate counterexample representation to the extended counterexample view, as this allows the programmer to decide which processing level they want to see the counterexample in.

Listing 3.2: Carbon's intermediate counterexample representation for verification error from the program in Listing 3.1.

```
        Intermediate Counterexample:
   Local Information:
Variable Name: null, Value: T@U!val!3, Type: Ref
Variable Name: this, Value: T@U!val!7, Type: Ref
Variable Name: s, Value: T@U!val!8, Type: Seq[Int]
T@U!val!8 with size 3 with entries:
 11 at index 0
   old Heap:
Heap entry: (T@U!val!7) + (T@U!val!1) --> (Value: T@U!val!23,
    Permission: 0/1)
Heap entry: (T@U!val!7) + (T@U!val!2) --> (Value: 12, Permission:
     0/1)
Heap entry: (T@U!val!3, T@U!val!12) + (T@U!val!7) --> (Value: T@U
    !val!26, Permission: 1/1)
   current Heap:
```

```
Heap entry: (T@U!val!7) + (T@U!val!1) --> (Value: 11, Permission:
    1/1)
Heap entry: (T@U!val!3, T@U!val!12) + (T@U!val!7) --> (Value: T@U
   !val!11, Permission: 0/1)
Heap entry: (T@U!val!7) + (T@U!val!2) --> (Value: 12, Permission:
    1/1)
```

Listing 3.3: Silicon's intermediate counterexample representation for verification error from the program in Listing 3.1.

```
      Intermediate Counterexample:
   Local Information:
Variable Name: null, Value: $Ref!val!1, Type: Ref
Variable Name: s, Value: Seq<Int>!val!1, Type: Seq[Int]
Variable Name: this, Value: $Ref!val!0, Type: Ref
Seq<Int>!val!0 with size 0 with entries:
Seq<Int>!val!1 with size 3 with entries:
 1 at index 0
   old Heap:
Heap entry: (StructA) + ($Ref!val!0) --> (Value: List(),
    Permission: 1/1)
   return Heap:
Heap entry: ($Ref!val!0) + (x) --> (Value: 1,
Permission: 1/1)
Heap entry: ($Ref!val!0) + (y) --> (Value: 2,
Permission: 1/1)
```

Listing 3.2 and 3.3 show us the generated intermediate counterexamples by the two backends, Carbon and Silicon, respectively, for the verification error from the program shown in Listing 3.1. The intermediate counterexample builds the structure for the extended counterexample. Still, it distinguishes itself by not evaluating the internal names to their names defined in the program. The most significant difference between both backends in their intermediate counterexample representation is their internal name generation. While Carbon uses the prefix "`T@U!val!`" concatenated with an integer for an internal value name independent of its type, Silicon provides different internal name structures such as "`$Ref!val!`" concatenated with an integer for its references, "`Seq<Int>!val!`" concatenated with an integer for sequences of integers or no internal name at all for fields and predicates. This leads to the intermediate counterexample format having a consistent structure across both backends. However, the value names are still bound to the backend-specific internal names.

Additionally, as shown in the Silicon counterexample in Listing 3.3, multiple sequences are defined even though the Viper program only defines one sequence. This can occur for every collection type in the intermediate counterexample representations of both backends. It happens when the model received by the SMT solver defines multiple instances of a collection type. We decided to present all defined instances of a collection type to give the

programmer as much internal information in the intermediate counterexample as possible.

In order to balance clarity and completeness in the representations of collections, we made a critical decision not to show collections as literals. Instead, we decided to list the elements of the collection that clearly specified in the counterexample model given by the SMT solver. This approach eliminates the issue of articulating extensive sequences or sets, in which only a subset of elements are specifically defined. A more thorough discussion on the representation of sequence/set/multiset and the reasons behind this choice will be covered in the following chapter.

Figure 3: The figure shows the structure of a Viper counterexample.

The primary structures of the common intermediate counterexample representation consist of the local information, the heaps, the functions, and the domains:

- The local information can be separated into two parts: the basic variables, such as integers and Booleans, and the collection types, such as sequences, sets, and multisets.
- Each heap belongs to a specific state in the program. The intermediate counterexample provides a view of each heap by showcasing:
  - Its fields with receivers, along with their permission amounts and values.
  - The predicates accessed inside of a heap, with details on their permissions.
  - Magic wands, including the variables and their corresponding values derived from the assertions.
- The intermediate counterexample also includes a view of the functions, clearly outlining those that are defined inside and outside of a user-defined domain.

3.2.2 Extended Counterexample

The `CounterexampleGenerator` class extends the intermediate counterexample. The generated counterexample can be accessed by executing the program verification with the flag "`--counterexample extended`."

The extended counterexample representation is generated after the intermediate representation is assembled and follows the same structure, independent from which backend is used. It is a human-readable form of the program state at a point where an assertion has failed and helps to understand the causes of the failure.

Listing 3.4: Carbon's extended counterexample representation for verification error from the program in Listing 3.1.

```
        Extended Counterexample:
   Store:
Collection variable "s" of type Seq[Int] with 3 entries:
 11 at index 0
Variable Name: this, Value: T@U!val!7, Type: Ref
   old Heap:
Predicate Entry: StructA(this) --> (Perm: 1/1)
Field Entry: this.y --> (Value: 12, Type: Int, Perm: 0/1)
Field Entry: this.x --> (Value: T@U!val!23, Type: Int, Perm: 0/1)
   current Heap:
Field Entry: this.y --> (Value: 12, Type: Int, Perm: 1/1)
Predicate Entry: StructA(this) --> (Perm: 0/1)
Field Entry: this.x --> (Value: 11, Type: Int, Perm: 1/1)
```

Listing 3.4 shows the extended counterexample for the verification error from the program shown in Listing 3.1. When comparing the extended counterexample from Listing 3.4 with the backend-specific intermediate counterexamples in Listing 3.2 and Listing 3.3, we can see that all internal names have been translated to the program-specific names and that heap entries are given a specific heap type.

When the heap does not uniquely map a specific identifier to a value, an internal identifier associated with a value may persist. To provide a specific example, consider a scenario where both `x` and `y` are assigned to the identifier `Ref!val!3`. In this case, we don't designate this identifier as `x` or `y`. Instead, the identifier `Ref!val!3` is left unchanged. The decision to keep the identifier `Ref!val!3` intact despite its presence in both `x` and `y` stems from the fact that replacing it could lead to misinterpretation. Therefore, certain identifiers remain untranslated under specific circumstances to maintain the integrity and clarity of the code.

Chapter 4

Added Features

Prior to this work, both Carbon and Silicon offered counterexample generation capabilities, but with different formats and encompassing varying elements of the Viper language. Hence, the goal of achieving interchangeability between the two backends is yet to be fulfilled. The limitations in the representation of counterexamples complicate the process of debugging program verification failures. This chapter introduces the newly added counterexample support of both backends for specific types in Viper.

In the following sections, the "initial implementation" or "original implementation" primarily refers (unless stated otherwise) to Silicon's previous counterexample generation, which is further described in Cedric Hegglin's [5] and Fabio Aliberti's [1] theses.

4.1 Basic Types

The representation of basic or primitive types, such as integers, Booleans, and references, was noticeably constrained in Viper's initial counterexample implementation. While a variable's actual value is usually captured, the representation lacks potentially relevant information for the programmer.

The new format has the internal name, the type, and the AST node, all of which were not present before. The internal name of variables is an essential information component since it provides the developer with a clear and direct relationship between the counterexample and the corresponding code in the original program. Moreover, a variable's type plays a significant part in future programming projects that build on top of the generated counterexample. Lastly, with the relation to the AST node, the programmer does not have to track where and how a variable was set. With these enhancements, the counterexamples are not only more comprehensible for programmers but also facilitate frontend code information extraction.

As a result, the internal name of the value assigned by the backend, the value's type, and the association of the value with its corresponding node in the Viper intermediate language were all incorporated in the intermediate and the extended counterexample version.

4.2 Collections

4.2.1 Sequences

While Silicon did support sequences in its original counterexample representation, Carbon did not. Nevertheless, there were several issues with the counterexample generation for sequences when using Silicon.

A key issue with Silicon's sequence generation is that it can sometimes yield sequences with elements based on default values. This issue stems from its dependence on SMT solvers, such as Z3, which often fill their models with default values.

For example, let us consider the Z3 model level function "Seq_index". If we request Z3 to generate a sequence of length three, with the specification that the first element must be 42, the function might return the sequence `[42, 42, 42]`. The "Seq_index" function's entries could then be presented as {`else -> 42`}, implying that for any index, the solver will provide 42 as the default value. However, this sequence could be misleading. The second and third elements were filled with default values and were not derived from explicit instructions in the code. Therefore, their actual values are undefined and could very well be different from 42. In this case, 42 might not be a relevant value for the second and third elements.

Although using these default values aids in providing a complete representation of sequences, it also introduces a risk of causing misconceptions about the actual values in the sequence. A different setting in Z3 allows for partial models, which yield sequences like [42, "#unspecified", "#unspecified"] due the "Seq_index" function's entries being {`0 -> 42, else -> #unspecified`}. Here, "#unspecified" stands for an undefined value, providing a clearer indication of unspecified sequence indices. An example showing this difference between the original and the new counterexample format is stated later in this chapter.

Recognizing these limitations, we introduced several improvements in this thesis, as can be seen in comparing the original and updated counterexamples from both Silicon and Carbon.

First, sequences were added to Carbon's counterexample support, unifying the representation across both backends.

Second, both backends' counterexample representations were expanded to provide more detailed information on each sequence, such as the specific type of the sequence and the internal name of values assigned by the backend.

Third, an issue with Silicon's original implementation was filling undetermined values in a sequence with a default value. This issue was addressed by utilizing a different Z3 setting – specifically, by requesting a partial model. As a result, unclear values are now marked as "#unspecified", providing an honest indication of the limitations of the counterexample.

Fourth, the new representation also changes how sequences are printed in the counterexample. Instead of displaying the entirety of a sequence, only the clearly determinable values are shown. For instance, consider a sequence of length 20,000. Presenting the entire sequence would be overwhelming for developers and not a practical method to express relevant information.

Listing 4.1: Viper program which contains sequences producing a verification error.

```
method update(values: Seq[Int], x: Int, y: Bool) returns (
    updatedValues: Seq[Int])
  requires |values| > 3
  requires values[0] == 2
  ensures  |values| == |updatedValues|
  ensures  updatedValues[0] != updatedValues[1]
  ensures  updatedValues[1] != updatedValues[2]
{
  updatedValues := values
  updatedValues := updatedValues[1 := 42]
  updatedValues := updatedValues[2 := 42]
}
```

Listing 4.2: Original counterexample generated by Silicon for the verification failure of the program shown in Listing 4.1.

```
    counterexample:
model at label old:
values <- (Seq<Int>!val!1): [2, 2, 2, 2]
x <- 0
y <- false
updatedValues <- (Seq<Int>!val!3): [2, 42, 42, 2]
on return:
values <- (Seq<Int>!val!1): [2, 2, 2, 2]
x <- 0
y <- false
updatedValues <- (Seq<Int>!val!3): [2, 42, 42, 2]
```

Listing 4.3: New counterexample generated by Carbon and Silicon for the verification failure of the program shown in Listing 4.1.

```
      Extended Counterexample:
   Store:
Collection variable "updatedValues" of type Seq[Int]
with 853 entries:
 2 at index 0
 42 at index 1
 42 at index 2
Variable Name: y, Value: #unspecified, Type: Bool
Variable Name: x, Value: #unspecified, Type: Int
Collection variable "values" of type Seq[Int] with 853 entries:
 2 at index 0
```

We will now illustrate the new representation using the example program with the method "`update`" shown in Listing 4.1. The method takes a sequence of integers that has at least length 3 and updates this sequence by replacing the first three elements with the integers `0`, `42`, and `42`, respectively. The postconditions ensure that the initial sequence and the updated sequence have the same size and that the first and second, and the second and third values of the updated sequence are not equal. As the method sets the second and third values of the updated sequence to `42`, the third postcondition is violated.

The original counterexample produced by Silicon shown in Listing 4.2 shows the values and sequences for the variables used in the method. The updated counterexample from the Silicon and Carbon backends shown in Listing 4.3 only lists the values of a sequence that could be determined from the partial model received from the SMT solver. The values for `x` and `y` are not specified in the counterexample, as they are irrelevant to the verification error. Thus, the counterexample given by the SMT solver does not contain any information about them.

4.2.2 Sets & Multisets

Prior to the extension of Viper's counterexample introduced with this thesis, Viper's counterexamples had a limited and problematic representation of sets and multisets. The Silicon backend was capable of handling sets, but neither backend could support multisets. Silicon's counterexample representation of sets was problematic, resembling the issues present in its representation of sequences. The counterexamples for sets showed values that might not be correct due to the partial models provided by SMT solvers. This could result in erroneous understandings of the sets included in the counterexample.

We addressed these problems by making several changes. Firstly, we integrated the ability to represent sets and multisets in Carbon's counterexam-

ples and added multisets to Silicon's counterexample features, thus, aligning the functionalities of both backends. Furthermore, Silicon's counterexample representation of sets was enhanced, in the same way as previously explained for its sequences, to provide more values (different from "`#unspecified`") for the contents of the sets. We achieved this by extracting more details from the internal functions of the counterexample model received from the SMT solver.

As not all values of the sets and multisets might be given in the counterexample model received by the SMT solver, the typical set representation format using "{"and "}" can lead to false visualizations of the sets and multistep. We addressed this problem in the updated counterexample format by listing all the identifiable elements of the set or multiset and stating its cardinality. This prevents the creation of inaccurate representations.

Listing 4.4: Viper program which contains sets producing a verification error.

```
method t2(a: Set[Int]) {
    var b: Set[Int] := Set(2)
    assert (a union b) == Set(1, 2)
}
```

Listing 4.5: Original counterexample generated by Silicon for the verification failure of the program shown in Listing 4.4.

```
    counterexample:
model at label old:
a <- (Set<Int>!val!1): {}
b <- (Set<Int>!val!0): {}
on return:
a <- (Set<Int>!val!1): {}
b <- (Set<Int>!val!0): {}
```

Listing 4.6: New counterexample generated by Carbon and Silicon for the verification failure of the program shown in Listing 4.4.

```
      Extended Counterexample:
   Store:
Collection variable "b" of type Set[Int] with 1 entries:
 2
Collection variable "a" of type Set[Int] with 0 entries:
```

The Viper program "`t2`" shown in Listing 4.4 takes a set of integers `a` as an argument. In this method, a new set `b` is declared and initialized with a set that only contains the integer `2`. The method then asserts that the union of `a` and `b` equals a set containing the integers `1` and `2`, which might be false in some cases and therefore fails verification. The original counterexample produced by Silicon shown in Listing 4.5 provides the internal

names `Set<Int>!val!1` and `Set<Int>!val!0` and states that both `a` and `b` are empty.

The updated counterexample, provides a much more detailed view. It lists the variable names, their types (i.e., `Set[Int]`), and the number of elements they contain. Additionally, it lists all the elements that could safely be determined in each set.

The improvements in the counterexample representation for sets are important because the counterexample model from the SMT solver might include only some values in the set or multiset. Hence, the previous approach using {} brackets and inserting the identifiable elements could lead to faulty counterexamples. By listing all identifiable elements and the cardinality of the set, the new counterexample format provides clear information for debugging.

4.3 Fields

The original counterexample representation of fields in Viper using the Silicon backend was already advanced compared to Carbon's initial counterexample support for fields, which did not show any heap information at all. Therefore, we aligned Carbon's counterexample generation for fields with Silicon's counterexample support and made minor improvements and adjustments to Silicon's counterexample representation of fields.

We implemented enhancements to the intermediate and extended counterexample representations to increase clarity for the verification failure. The revised representations include the permission, the value, and the assigned value type.

In the extended counterexample, the internal identifier of references are only translated to real names when they occur once, mitigating potential confusion when different references share the same internal names in the implementation. To illustrate this, let us consider a Viper program that includes three reference arguments `this`, `that` and `other`. When Silicon verifies the program, the Z3 counterexample model may assign internal identifiers `$Ref!val!0`, `$Ref!val!0` and `$Ref!val!1` to these three references respectively. This assignment can occur when `this` and `that` are semantically equivalent, leading to the same internal identifier. In this case, the extended counterexample would not translate `$Ref!val!0` back to either `this` or `that`. However, it would translate `$Ref!val!1` back to its real name `other`.

Beyond the printed representation, we made certain information accessible to programmers, such as grouping a field with the node it gets assigned to in the program. This can help programmers work with the counterexample on a code level. Coherent with counterexample values of other types,

another modification involved marking field values as "#unspecified" when their values could not be reliably determined, rather than using a default value from the counterexample model by the SMT solver as was the case in Silicon's original representation.

The adjusted format of Silicon's previous counterexample representation of fields is the new counterexample format of fields that both Carbon and Silicon use.

Listing 4.7: Viper program which contains fields producing a verification error.

```
field next: Bool

method foo(x: Ref) returns (value: Bool)
  requires acc(x.next) && x.next
  ensures value != true
{
  value := x.next
  x.next := false
  assert !value
}
```

Listing 4.8: Original counterexample generated by Silicon for the verification failure of the program shown in Listing 4.7.

```
    counterexample:
model at label old:
x <- Ref ($Ref!val!0) {
  next(perm: 1/1) <- true
}
value <- true
on return:
x <- Ref ($Ref!val!0) {
  next(perm: 1/1) <- false
}
```

Listing 4.9: New counterexample generated by Carbon and Silicon for the verification failure of the program shown in Listing 4.7.

```
      Extended Counterexample:
   Store:
Variable Name: value, Value: true, Type: Bool
Variable Name: x, Value: T@U!val!6, Type: Ref
   old Heap:
Field Entry: x.next --> (Value: true, Type: Bool, Perm: 1/1)
   current Heap:
Field Entry: x.next --> (Value: false, Type: Bool, Perm: 1/1)
```

The Viper program shown in Listing 4.7 consists of a field `next` of type Boolean and a method "`foo`" that returns a Boolean `value`. The method

takes as input a reference `x`, requiring that we have access to the `next` field of `x` and that `x.next` is true. The method sets `value` to `x.next` and then sets `x.next` to false. The method also contains an assertion stating the negation of `value` that value is true, which leads to a verification error.

The original counterexample generated by Silicon shown in Listing 4.8 presents the values and permissions of `x.next` in the `old` and `current` states.

The updated counterexample shown in Listing 4.9 displays the state of the variables in the `Store` section. In each heap section, the value, type, and permission of a field access are denoted.

4.4 Predicates

Neither backend presents predicates in their original counterexample representations. As a result, programmers neither see to which predicates permissions are held nor obtain any information about the values of the heap locations contained within the predicate.

The new counterexample implementation for both backends displays all predicates in a heap state along with their permissions. It also shows the contents of folded predicates with its assigned values. Additionally, it also provides additional information to the programmer through the intermediate counterexample, thus offering greater insight into the predicate.

Listing 4.10: Viper program which contains fields and predicates producing a verification error.

```
field left: Int
field right: Int

predicate tuple(this: Ref) {
    acc(this.left) && acc(this.right)
}

method setTuple(this: Ref, l: Int, r: Int)
    requires tuple(this)
    ensures tuple(this) && (unfolding tuple(this) in this.left +
    this.right) == old(unfolding tuple(this) in this.left + this.
    right + 1)
{
    unfold tuple(this)
    this.left := l
    this.right := r
    fold tuple(this)
}
```

Listing 4.11: Original counterexample generated by Silicon for the verification failure of the program shown in Listing 4.10.

```
    counterexample:
model at label old:
this <- Ref ($Ref!val!0) {
}
l <- 0
r <- 0
on return:
this <- Ref ($Ref!val!0) {
}
l <- 0
r <- 0
```

Listing 4.12: New counterexample generated by Carbon and Silicon for the verification failure of the program shown in Listing 4.10.

```
      Extended Counterexample:
   Store:
Variable Name: r, Value: (- 669), Type: Int
Variable Name: l, Value: 176, Type: Int
Variable Name: this, Value: T@U!val!7, Type: Ref
   old Heap:
Predicate Entry: tuple(this) --> (Perm: 1/1) {
   this.left --> 264
   this.right --> (- 864)
}
Field Entry: this.left --> (Value: 264, Type: Int, Perm: 0/1)
Field Entry: this.right --> (Value: (- 864), Type: Int, Perm:
    0/1)
   current Heap:
Predicate Entry: tuple(this) --> (Perm: 1/1) {
   this.left --> 176
   this.right --> (- 669)
}
Field Entry: this.left --> (Value: 176, Type: Int, Perm: 0/1)
Field Entry: this.right --> (Value: (- 669), Type: Int, Perm:
    0/1)
```

The Viper program shown in Listing 4.10 defines a field `left` and `right` of type integer, and a predicate `tuple` that represents the ownership of these fields. The method "`setTuple`" requires the `tuple` predicate with an input `this` (of type reference), and it updates the `left` and `right` fields of `this` with the given integer variables `l` and `r`, respectively. It ensures that after the update, the sum of the `left` and `right` fields equals the old sum plus one, which might not be true. Thus, verification fails. The original Silicon counterexample shown in Listing 4.11 presents the integer values of the variables `l` and `r`. However, it does not provide any information on the `tuple(this)` predicate instance. We can only see the internal name of the reference `this`. The updated counterexample for Carbon and Silicon shown

in Listing 4.12, provides more detailed information. It adds information about the permissions to the `tuple` predicate in the `old` and `current` heap. Additionally, it presents the values of the two field accesses inside the folded predicate `tuple`. Therefore, we can see that `this.left` and `this.right` have the same values in the "current" heap as `l` and `r`, respectively.

4.5 Magic Wands

Neither backend supports magic wands in their counterexample representations.

Listing 4.14: Viper program which contains a magic wand producing a verification error.

```
field f: Int
field g: Int
predicate P(r: Ref) { acc(r.f) }

method m1(x: Ref)
  requires acc(x.f) --* P(x)
{
  assert acc(x.g) --* P(x)
}
```

Listing 4.15: New counterexample generated by Carbon and Silicon for the verification failure of the program shown in Listing 4.14.

```
      Extended Counterexample:
   Store:
Variable Name: x, Value: $Ref!val!0, Type: Ref
   old Heap:
Magic Wand Entry: wand@0 --> (Left: acc(x.f, 1/1), Right: acc(P(x
    ), 1/1), Perm: 1/1)
   current Heap:
Magic Wand Entry: wand@0 --> (Left: acc(x.f, 1/1), Right: acc(P(x
    ), 1/1), Perm: 1/1)
```

Consider the Viper program shown in Listing 4.14, which fails verification due to the usage of a nonexistent magic wand. The new counterexample for both backends shown in Listing 4.15 clarifies that we only have permission for a magic wand using `x.f` as field access.

4.6 Quantified Permissions

Quantified permissions offer the advantage of providing point-wise specifications instead of recursive ones. They play a crucial role in specifying unbounded heap structures in Viper.

In the original representations of both Carbon and Silicon, no support was implemented for quantified permissions of heap resources in counterexamples. We identified that, similar to sequences where not all values might be retrievable, we faced incomplete information regarding the heap resources that quantified permissions accessed. This incompleteness in the counterexample can lead to the misconception that certain heap resources, to which permissions might be granted, are absent, even though they might be present in the underlying program.

To address this, the generated counterexample explicitly states that permission to other heap resources might be granted when quantified permissions occur in the verified program. Besides that, quantified permissions related to heap resources, such as fields, predicates, or magic wands, are represented in the counterexample similarly to their non-quantified versions.

4.7 Functions & Domains

In Chapter 2.4, we mentioned that the current support of functions, both those defined inside and outside of user-defined domains (the latter known as heap-dependent functions), in Silicon's counterexamples was already advanced. Some minor adjustments to the representation of functions and domains had to be made such that the format could be coherent with the newly added representation of functions and domains in Carbon's counterexamples.

The objective of the new counterexample format tries to give the same counterexample for both backends. However, minor differences might still arise, primarily in the way heap identifiers are translated in heap-dependent functions. As their name suggests, heap-dependent functions rely on specific instances of the heap. Therefore, it is necessary to establish a mechanism that assigns unique names to each heap instance. Carbon converts heap identifiers to heap state labels when possible, and retains the heap identifier when it is not. This process is based on Silicon's original method of function extraction, but in Silicon every heap identifier is converted to a heap state label.

The updated representation of the counterexample aims to maximize the independence from the backend, resulting in a more coherent presentation.

Listing 4.16: Viper program which contains domains and functions producing a verification error.

```
field f: Int

domain List[T] {
    function nil(): List[T]
    function cons(x: T, xs: List[T]): List[T]
    axiom nil_cons {
        forall z: T, zs: List[T] :: cons(z, zs) != nil()
    }
}

function foo(r: Ref): Int
  requires acc(r.f)
{
  r.f
}

method test(x: Ref, xs: List[Int], n: List[Int])
  requires n == nil()
  requires acc(x.f)
{
  assert n != cons(5, xs)
  assert foo(x) == 5
}
```

Listing 4.17: Original counterexample generated by Silicon for the verification failure of the program shown in Listing 4.13.

```
    counterexample:
model at label old:
x <- Ref ($Ref!val!0) {
  f(perm: 1/1) <- 0
}
xs <- List[Int]_1 where {

  }
n <- List[Int]_0 where {

  }
on return:
x <- Ref ($Ref!val!0) {
  f(perm: 1/1) <- 0
}
xs <- List[Int]_1 where {

  }
n <- List[Int]_0 where {

  }
Domain:
domain List[Int]{
 nil{
```

```
    List[Int]_0
}
cons{
    List[Int]_2
}
}
Functions:
foo{
    0
}
```

Listing 4.18: New counterexample generated by Carbon and Silicon for the verification failure of the program shown in Listing 4.13.

```
      Extended Counterexample:
   Store:
Variable Name: n, Value: T@U!val!11, Type: List[Int]
Variable Name: xs, Value: T@U!val!8, Type: List[Int]
Variable Name: x, Value: T@U!val!7, Type: Ref
   old Heap:
Field Entry: x.f --> (Value: 10, Type: Int, Perm: 1/1)
   current Heap:
Field Entry: x.f --> (Value: 10, Type: Int, Perm: 1/1)
   Domains:
domain List[Int]{
 nil():List[T]{
    T@T!val!2 -> n
    else -> #unspecified
}
cons(T,List[T]):List[T]{
    5 xs -> T@U!val!10
    else -> #unspecified
}
}
foo(Ref):Int{
    Heap@@13 x -> 10
    else -> #unspecified
}
```

The Viper program shown in Listing 4.16 declares a generic domain `List[T]`, which is essentially a linked list structure with two functions: `nil` constructs an empty list, and `cons` builds a non-empty list by prepending an element to an existing list. The `nil_cons` axiom states that a non-empty list is not equal to an empty list. Additionally, a non-domain function foo returns the integer responding to the field access `r.f`, and a method "`test`" checks if a fresh list `n` equals a list constructed by `cons`.

The original Silicon counterexample shown in Listing 4.17 demonstrates the failure of the assertion `foo(x) == 5` in the "`test`" method. This is shown through the counterexample model (which presents the state of the variables at the point of failure), where the integer of the field access `x.f` is `0`. The

terms `List[Int]_0` and `List[Int]_2` represent specific list instances. The function `foo` always returns `x.f`, which is `0` in this case.

In the updated Silicon and Carbon counterexample, each instance of the `List[Int]` domain is associated with a unique identifier in the Domains section. The heap state is stated as the first parameter for the function `foo`. For any other input or heap state, the function results in "`#unspecified`".

Chapter 5

Information Extraction

In this chapter, we will discuss the technical details of the feature extraction from the counterexample model received from the SMT solver. We will split this chapter into three parts: First, explaining the extraction of basic types and collection types, such as sequences or sets. Secondly, we will focus on the extraction of heap information. This includes extracting information about heap resources, such as fields, predicates, or magic wands. Lastly, we will explain the counterexample generation of functions and domains.

5.1 Basic & Collection Types

The following section explains how basic and collection types are retrieved from the counterexample model provided by the SMT solver in both backends. We will demonstrate this process on the program shown in Listing 5.1.1. The program consists of a method that accepts a Boolean `b`, an integer `i`, and a set of integers `n` as input. It has three preconditions: `i` must be less than `6`, `b` must be `true`, and `n` must be an empty set. The method begins by initializing `n` with an empty set of integers. The first assertion checks that `b` is `true`, and the second assertion checks whether a set containing only `i` equals `n`. Since `n` is an empty set, the second assertion fails. Thus, in the ideal counterexample, we would like to obtain counterexample values for the integer `i`, the Boolean `b`, and the set of integers `n`.

Retrieving basic types from the counterexample model works similarly in both backends. Basic types, like integers, Booleans, or references, can be directly accessed in the counterexample model once their internal identifiers are known.

Listing 5.1.1: Viper program containing an integer, a Boolean and a set of integers.

```
method basicMethod(b: Bool, i: Int, n: Set[Int])
  requires i < 6
  requires b == true
  requires n == Set()
{
  assert b
  assert Set(i) == n
}
```

In Carbon, the internal identifier of a value consistently follows a structure consisting of either the value's name alone or the value's name accompanied by an underscore or an `@` symbol, followed by an integer. The reasoning behind this structure can be better understood when considering the transformation from Viper to Boogie, especially in the context of the SSA (Static Single Assignment) form. SSA form ensures that each value is assigned exactly once. In Viper, subsequent assignments to a value, such as {`x:=0; x:=1`} are translated in Boogie to {`x@0:=0; x@1:=1`}. This transformation highlights why Carbon employs the described internal identifier structure. If a value has multiple internal identifier instances in the model, the latest instance is determined by the integer in the internal identifier: the larger the integer, the more recent is the value instance. Thus, to look up the value of a value in the counterexample model, we use the internal identifier with the highest integer. Listing 5.1.2 shows the basic values' internal identifiers and how they map to their values as found in the model provided by the SMT solver.

Listing 5.1.2: Mapping of Carbon internal value names to their counterexample values.

```
b_2 -> true
i -> (- 900)
```

In Silicon, we directly get the symbolic value, which we refer to as internal identifier of a value in the `Store`. The `Store` maps the program's value names to the internal identifiers from the model. Listing 5.1.3 shows the mapping from the internal value name to its corresponding counterexample value as represented in the counterexample model.

Listing 5.1.3: Mapping of Silicon internal value names to their counterexample values.

```
b@3@04 -> true
i@4@04 -> 0
```

For collection types, such as sequences, sets, or multisets, we first determine the internal identifiers of the values, as previously explained for both

backends. Next, we extract their content by compiling data from specific functions in the counterexample model associated with each collection type.

Listing 5.1.4: Internal functions defining the counterexample content of a set in Carbon.

```
n -> T@U!val!6
Set#Empty -> {
    T@T!val!2 -> T@U!val!6
    else -> #unspecified
}
```

For the method "basicMethod", the functions as they would appear in the Carbon backend's counterexample model are illustrated in Listing 5.1.4. The internal functions have the same structure in Silicon, besides their names, which change slightly.

Since the variable n in the program is only defined as an empty set, we can gather all the information for our set through the internal function that defines empty sets. This function maps the identifier of the set type (in this case integers) to the identifier of the corresponding empty set.

5.2 Heap Resources

In this section, we must distinguish how heap information is extracted for the two backends. The counterexample heap models received from the SMT solver significantly differ for both backends. Hence, we will begin by focusing on extracting heap resources in Carbon and later in Silicon.

To understand how the heap features are extracted, we will illustrate the information extraction process using a Viper program shown in Listing 5.2.1. The program includes a field, a predicate, and a magic wand.

Listing 5.2.1: Viper program failing verification due to wrong wand arguments in the second assertion.

```
field x: Int
field y: Int
field next: Int

predicate Struct(this: Ref) {
  acc(this.x) && acc(this.y)
}

method foo(a: Ref, b: Ref)
  requires acc(a.next) --* Struct(b)
  requires acc(a.next)
  requires Struct(b)
{
  a.next := 5
```

```
  unfold Struct(b)
  b.x := 7
  b.y := 11
  fold Struct(b)
  assert acc(a.next) --* Struct(b)
  assert acc(b.next) --* Struct(b)
}
```

5.2.1 Carbon Heap Extraction

In Carbon, each heap instance consists of two components - a heap state and a mask state. For example, in the program shown in Listing 5.2.1, the "`old`" label (which signifies the initial instance of the heap for the "`foo`" method) consists of a heap state identifier, `Heap@@10 -> T@U!val!9`, and a mask state identifier, `Mask@@9 -> T@U!val!18`. These different heap and mask state identifiers can be identified within the counterexample model received from the SMT solver. The first heap and mask states are internally named, beginning with `Heap@@` or `Mask@@`, succeeded by a number. This forms the starting point for the different heap instances of the counterexample.

Listing 5.2.1.1: Mapping showing the order of heap states.

```
succHeap -> {
    T@U!val!13 T@U!val!15 -> true
    T@U!val!8 T@U!val!13 -> true
    T@U!val!9 T@U!val!8 -> true
    else -> #unspecified
}
```

Listing 5.2.1.2: Mapping showing the combinations of heap states and mask states resulting in one heap instance of the program.

```
state -> {
    T@U!val!13 T@U!val!17 -> true
    T@U!val!9 T@U!val!18 -> true
    T@U!val!8 T@U!val!17 -> true
    [...]
    else -> #unspecified
}
```

We use the function "`succHeap`", as shown in Listing 5.2.1.1, to determine the occurrence order of heap instances. This function contains a mapping where the second entry represents the succeeding heap state identifier to the first entry (also representing a heap state identifier). The corresponding mask state for each heap state can be determined using the "`state`" function from the model, shown in Listing 5.2.1.2. Each mapping corresponds to one heap instance in this function, represented by the first (a heap state identifier) and second entry (a mask state identifier) in a specific mapping.

After identifying all heap instances, we must now determine the heap resources and their associated permissions for each heap instance. To achieve this, we utilize the "`MapType0Select`" and "`MapType1Select`" functions. Specifically, the "`MapType0Select`" function serves as the map lookup for heap values, while "`MapType1Select`" is used for looking up permissions, given that both heap and mask are structured as maps.

Field Accesses We will now see how to determine the value and permission of a specific field access within a heap instance. To do so, we will use the example program detailed in Listing 5.2.1. Within the model's function titled "`MapType0Select`", there is a mapping `T@U!val!14 T@U!val!7 T@U!val!1 -> 5`. The sequence key's first input refers to the heap state, the second represents the reference, and the third identifies the specific field being accessed. The output ("`5`") signifies the value assigned to that field access. To determine the permission of the field access, we refer to another mapping titled "`MapType1Select`". We look for a key that combines the mask state identifier (which, in conjunction with the heap state identifier `T@U!val!14`, constitutes a heap instance) and the identifiers `T@U!val!7` and `T@U!val!1`. This key reveals the specific field access's permission.

In the counterexample provided by Z3, the "`MapType_Select`" function only contains entries for each heap instance that were explicitly referenced in the program. Consequently, Carbon separates the changes of heap resources across different heap instances. To gather the heap resources that have not been assigned, it is necessary to backtrack through the heaps sequentially. During this backtracking, the states of heap resources are incorporated into the heap state only if a state for that particular heap resource hasn't already been included.

To demonstrate this, consider the Viper program shown in Listing 5.2.1.3. The program compares two integers accessed via `this.first` and `this.second`. As these were previously assigned the integers `1` and `2`, the assertion that these two are equal is incorrect. To determine the counterexample for the heap at the point of failure, it is necessary to backtrace through all heap instances. This is because the "`MapType_Select`" function in the counterexample received from the SMT solver only states that we have full permission for the field access `this.second`, and that `this.second` is `2` for the last heap instance. By incorporating the previous heap instance, we get additional information from the "`MapType_Select`" function that we also have full permission for the field access `this.first` and that `this.first` is `1`. This information is essential for a complete counterexample.

Listing 5.2.1.3: Viper program failing verification due to wrong assertion.

```
field first: Int
field second: Int

method foo(this: Ref)
    requires acc(this.first)
    requires acc(this.second)
{
    this.first := 1
    this.second := 2
    assert this.first == this.second
}
```

Predicate Accesses We determine predicate accesses in a manner similar to how we approach field accesses. The only distinction is that, in the input sequences of the "`MapType0Select`" and "`MapType1Select`" functions, the second identifier always signifies `null`. The third identifier consequently represents the actual predicate.

To determine that the mappings within the functions indicate a predicate access, we refer to the "`IsPredicateField`" function, as demonstrated in Listing 5.2.1.3. If the resource identifier is a predicate identifier, this mapping will assign 'true' to it.

Listing 5.2.1.4: Mapping showing the identifiers that represent a predicate.

```
IsPredicateField -> {
    T@U!val!1 -> false
    T@U!val!11 -> true
    T@U!val!15 -> false
    T@U!val!17 -> false
    else -> #unspecified
}
```

Every predicate has its own function within the counterexample model provided by the SMT solver. These function names are derived from the respective predicate names. The methodology used to generate these function names is the same as the one previously detailed for creating internal identifiers for values. For instance, in the Viper program presented in Listing 5.2.1, the function for the predicate named `Struct` is simply named after the predicate itself. This function can be seen in Listing 5.2.1.4. These specific functions perform a single mapping: They map the internal identifiers of the arguments associated with a predicate to the predicate's own identifier.

Listing 5.2.1.5: Mapping from the identifiers of the arguments the `Struct` predicate has to the predicate identifier.

```
Struct -> {
    T@U!val!6 -> T@U!val!11
    else -> #unspecified
}
```

To determine the counterexample state of a predicate we back-translate the internal identifiers of the arguments that were previously identified. The predicate identifier, along with the "`MapType1Select`" function, helps us figure out the permission we hold for a specific instance of the predicate in the mask. Additionally, using this predicate identifier and the "`MapType0Select`" function, we can ascertain the value assigned to the predicate in the heap. This heap value represents the values of the heap resources enclosed within the predicate, provided it has ever been folded or unfolded. It corresponds to a frame, which can consist of one part or multiple parts combined using the "`CombineFrames`" function. Each part is either an empty frame or a specific frame fragment. For example, for a predicate with `acc(this.x) && acc(this.y)` as a body (where this is a reference and x and y are fields), it would resemble `CombineFrames(FrameFragment(v1), FrameFragment(v2))`. However, since we currently only have the name of the value, it is necessary to determine how it was constructed.

We now examine the "`MapType0Select`" function. Each mapping inside the function points to an identifier, which can used to determine the values inside of a predicate. To do so, we recursively traverse through the "`CombineFrames`" function within the counterexample model. We lookup which sequence consisting of two identifiers points to the identifier identifying the inside of the predicate. Then, we save the first identifier of the sequence and we recursively do the same for the second identifier in the sequence until no sequence points to the current identifier we want to lookup. The resulting sequence of identifiers denotes the frames for the values inside of a predicate.

For example, the "`CombineFrames`" function for the program in Listing 5.2.1 is shown in Listing 5.2.1.5. For the predicate access `Struct(b)` we start the recursive lookup with the identifier `T@U!val!38`. This gives us the sequence of identifiers consisting of `T@U!val!35` and `T@U!val!37`. Since we can't find another sequence for the identifier `T@U!val!37`, we stop the recursive lookup. Thus, the final combination of frame identifiers for the values inside of the predicate `Struct(b)` is `T@U!val!35` and `T@U!val!37`.

Listing 5.2.1.6: Mapping from sequences of two identifiers to a single identifier.

```
CombineFrames -> {
    T@U!val!19 T@U!val!21 -> T@U!val!22
    T@U!val!35 T@U!val!37 -> T@U!val!38
    else -> #unspecified
}
```

We can lookup the value of a certain frame in the "`FrameFragment`" function within the counterexample model provided by the SMT solver. The "FrameFragment" function for the program presented in Listing 5.2.1 is shown in Listing 5.2.1.6. As previously determined, the first and second frame of the predicate access `Struct(b)` are assigned to the identifiers `T@U!val!35` and `T@U!val!37`, respectively. Therefore, we can determine that the first frame has value `7` and the second frame has the value `11`.

Finally, we can compare the structure of a predicate to the frames. From the program in Listing 5.2.1, it is easily visible that the structure `acc(this.x) && acc(this.y)` of the predicate `Struct` consists of two fields. Thus, we assign the value `7` of the first frame to `this.x` and the value `11` of the second frame to `this.y`.

Listing 5.2.1.7: Mapping from the values to a specific frame identifying the values of the arguments contained inside a predicate.

```
FrameFragment -> {
    11 -> T@U!val!37
    7 -> T@U!val!35
    T@U!val!18 -> T@U!val!19
    T@U!val!20 -> T@U!val!21
    else -> #unspecified
}
```

Magic Wands The process of determining magic wands resembles the procedure for establishing predicate accesses, with one key difference: the third identifier now signifies the magic wand identifier. To ascertain whether the resource identifier is a magic wand, we look at the "`IsWandField`" function, similar to what we did with predicates. We need to use this identifier to locate the arguments involved in the assertion of the magic wand. As illustrated in Listing 5.2.1.4, the input sequence of the mapping consists of the values and permissions utilized in the magic wand's assertion. The output represents the identifier from the utilized magic wand. Translating these values provides us with the information for the final counterexample representation of the magic wand.

Listing 5.2.1.8: Mapping showing the mapping from the values used in the assertion to the magic wand identifier.

```
wand -> {
    T@U!val!6 1.0 T@U!val!6 1.0 -> T@U!val!17
    T@U!val!7 1.0 T@U!val!6 1.0 -> T@U!val!15
    else -> #unspecified
}
```

Quantified Permissions Determining the values and permissions of heap resources with quantified permissions does not differ from the previously explained process. The only variation is that when looking up the permission in the "`MapType1Select`" function, the mask identifiers used are labeled as quantified permission masks.

5.2.2 Silicon Heap Extraction

In Silicon, each heap instance consists of a list of chunks. A chunk can represent field accesses, predicate, magic wands, or heap resources with quantified permissions.

Field Chunk In a field chunk, the accessed field's name is directly given. Therefore, we only need to evaluate the given expressions in the chunk to receive the internal name of the receiver and the value assigned to the field reference. Details on how to evaluate both of these expressions can be found in chapter 4.1 of Cedric Hegglin's Thesis, "Counterexample for a Rust Verifier" [5]. The permission accesses can be determined similarly.

Predicate Chunk The name of a predicate is directly given inside its heap chunk. The internal names of the predicate arguments can be determined by independently evaluating the expressions given in the chunk. The permissions can, again, like for field chunks, be determined by evaluating the perm expression of a field chunk.

To ascertain the values contained within a predicate, we evaluate its "`snapshot`". This "`snapshot`" follows the same structure as "`frame combinations`" in Carbon. The keyword `$Snap.combine`, similar to the "`CombineFrames`" function in Carbon, combines the different snapshot arguments. In Silicon, the Carbon equivalent of an empty frame is denoted as "`$Snap.unit`". A "`snapshot`" can be evaluated by employing the "`$SortWrappers._To$Snap`" functions found in the counterexample model received from the SMT solver. These functions are comparable to "`FrameFragment`" function in Carbon.

Silicon's "`snapshot`" can consist of values and terms. As an example, a possible term inside of a "`snapshot`" could be "`Second:(Second:($t@4@04))`".

When looking up the identifier `$t@4@04` in the model received from the SMT solver, we will get `($Snap.combine $Snap.unit ($Snap.combine $Snap.unit ($Snap.combine $Snap.unit $Snap.unit)))`. Thus, from "`Second:(Second:($t@4@04))`", we know that we only need to evaluate the second "`snapshot`" identifier of the second "`$Snap.combine`". This leaves us with the "`$Snap.unit`" identifier. We then lookup the value for "`$Snap.unit`" within the model, which gives us the value of the inside of a predicate.

Wand Chunk As discussed earlier, when retrieving magic wands within Carbon, we evaluate the values present within the two assertions of a magic wand. This involves examining each value within both assertions and evaluating them. This evaluation is based on the same processes as previously defined for the other counterexample types.

Quantified Permissions The extraction process for heap resources with quantified permissions is similar to that of their non-quantified counterparts, with a slight difference in the evaluation of permissions.

For quantified fields, we have specific functions within the counterexample model produced by the SMT model. Each function's name is the combination of "`$FVF.lookup_`" and the field's name. This function is a mapping pointing from an identifier denoting the instance of the quantified field and the identifier denoting the reference to the value of the quantified field access. To determine the permission, we evaluate the quantified permission of the field for the previously determined reference used in the field access.

A quantified predicate is determined through its permissions. That means, that we fist evaluate the quantified permissions for each combination of reference arguments of the quantified predicate and then add all the predicate instances corresponding to specific argument combinations which have a valid permission.

As an example, we consider a predicate `P1` with the quantified permission "`requires forall i: Int :: 0 <= i && i < 10 ==> P1(i)`". The goal is to identify all potential predicate arguments that satisfy the condition "`0 <= inv@0@1(i) && inv@0@2(i) < 10`" specified in the quantified predicate chunk. The permission term includes two inverse functions: "`inv@0@1(i)`" and "`inv@0@2(i)`". Each of these inverse functions encompasses several possible arguments for the permission term. Therefore, it is necessary to check the validity of the permission term across all potential combinations of arguments from the two inverse functions. After identifying the argument combinations that yield a valid permission term, we narrow down to the subset where both arguments are identical, as both inverse functions must represent the same argument for the variable `i`. This subset of arguments

for the predicate `P1` reveals all possible quantified predicate instances that are relevant for the program that fails verification.

5.3 Functions & Domains

We will now describe the process of extracting functions and domains from the counterexample model generated by the SMT solver. To better understand this extraction process, we will begin by discussing how domains and functions are represented.

The domain is represented by its type, followed by a listing of the domain-functions instances in the program. Functions and domain-functions in the counterexample model describe the mappings of inputs to outputs. In both Carbon and Silicon, functions are described through a set of options. Each option in the function definition corresponds to a specific sequence of arguments and defines the value the function returns for that input. Additionally, default cases exist for handling inputs not covered by any option, but due to the partial model of the SMT solver, the option is marked as "#unspecified".

To illustrate the extraction process, let's consider a program example involving a domain called "`List[T]`" and a function "`foo`" as shown in Listing 5.3.1.

Listing 5.3.1: Viper program consistings of domains and functions.

```
field f: Int

domain List[T] {
    function nil(): List[T]
    function cons(x: T, xs: List[T]): List[T]
    axiom nil_cons {
        forall z: T, zs: List[T] :: cons(z, zs) != nil()
    }
}

function foo(r: Ref): Int
  requires acc(r.f)
{
  r.f
}

method test(x: Ref, xs: List[Int], n: List[Int])
  requires n == nil()
  requires acc(x.f)
{
  assert n != cons(5, xs)
  assert foo(x) == 5
}
```

Listing 5.3.2: Information received from the SMT solver in Carbon regarding the functions and domains.

```
Heap@@13 -> T@U!val!4
n -> T@U!val!8
xs_1 -> T@U!val!5
foo -> {
    T@U!val!4 10 -> 12
    else -> #unspecified
}
nil -> {
    T@T!val!2 -> T@U!val!8
    else -> #unspecified
}
cons -> {
    5 T@U!val!5 -> T@U!val!7
    else -> #unspecified
}
```

Listing 5.3.3: Information received from the SMT solver in Silicon regarding the functions and domains.

```
foo -> {
    $Snap.unit 10 -> 12
    else -> #unspecified
}
cons<List<Int>> -> {
    5 List<Int>!val!1 -> List<Int>!val!2
    else -> #unspecified
}
nil<List<Int>> -> List<Int>!val!0
```

In Listings 5.3.2 and 5.3.3, we have mappings for the function "`foo`": "`T@U!val!4 10`" for Carbon and "`$Snap.unit 10`" for Silicon, both of which point to the integer `12`. Since "`foo`" is a heap-dependent function, the first argument is an identifier for the heap instance, while the second one signifies the input argument of the function for the reference `r`. The output of the "`foo`" function for this argument in this heap instance is the integer `12`, as indicated by the mapping. For all other inputs, the function's output is "`#unspecified`".

The counterexample for the "`cons`" function is determined similarly. The key distinction is that the first argument in the mapping does not symbolize the heap instance but directly represents the function's first argument. Consequently, the mappings in the counterexample received from the SMT solver for each backend comprise two arguments pointing to another one. It is evident that the only feasible arguments are "`5 T@U!val!5`" for Carbon and "`5 List<Int>!val!1`" for Silicon, where the two arguments denote the integer and the identifier for a `List[Int]` value that the "`cons`" function accepts as input. The result is then an identifier for a variable of type `List[Int]`.

The procedure for extracting counterexample information for the "nil" function is the same as for the "cons" function. After extracting the counterexample information for all three functions as described and translating all identifiers to their actual values, we obtain the counterexample shown in Listing 5.3.4 for the program in Listing 5.3.1.

Listing 5.3.4: Counterexample produced in Silicon for the program failing verification shown in Listing 5.3.1.

```
    counterexample:
      Extended Counterexample:
   Store:
Variable Name: n, Value: List<Int>!val!0, Type: List[Int]
Variable Name: xs, Value: List<Int>!val!1, Type: List[Int]
Variable Name: x, Value: $Ref!val!0, Type: Ref
   current Heap:
Field Entry: x.f --> (Value: 0, Type: Int, Perm: 1/1)
   old Heap:
Field Entry: x.f --> (Value: 0, Type: Int, Perm: 1/1)
   Domains:
domain List[Int]{
 nil{
    List<Int>!val!0
}
cons(T,List[T]):List[T]{
    List<Int>!val!1 -> List<Int>!val!2
    else -> #unspecified
}
}
foo(Ref):Int{
    Heap@0 x -> 0
    else -> #unspecified
}
```

5.4 Limitations

Unfortunately, there certain limitations in the previously described generation of counterexamples:

- For both backends generating counterexamples for predicate snapshots involves comparing the snapshot to the actual predicate body. However, the predicate body can be exceedingly complex, making the comparison sometimes unfeasible. As an example, the comparison is not possible when the predicate body references another predicate. In such scenarios, the counterexample will label the predicate body as "unspecified".
- In Silicon, generating counterexamples for quantified permissions requires evaluating the permission term for various arguments. How-

ever, this permission term may include expressions that are currently unsupported, such as conditional expressions (i.e., exp1 ? exp2 : exp3).

Chapter 6

Evaluation

In order to determine the effectiveness of newly implemented features, we performed a number of experiments evaluating the runtime impact of our counterexample generation process. Specifically, we compared the runtime of tests conducted without counterexample generation to those with the inclusion of the intermediate counterexample and the extended counterexample, separately. We evaluate the produced overhead for Silicon and Carbon, resulting in a total of four quantitative analyses. Figures 6.1, 6.2, 6.3, and 6.4 illustrate our results.

The tests used for our comparative study comprised 883 tests from the Viper test suite. It is important to note that all these tests failed verification due to various Viper language features and that we ran each test twice, taking the mean runtime as a result to reduce inconsistencies in the results.

All our evaluations were conducted on a specific hardware setup to ensure uniformity. The tests were performed on a MacBook Pro, with the following specifications: Model Identifier - MacBookPro 14,2; Processor - Dual-Core Intel Core i5 operating at 3.1 GHz; Memory - 8 GB; Total Number of Cores - 2, with L2 Cache of 256 KB per core and L3 Cache of 4 MB; Hyper-Threading Technology was enabled. It's essential to note that the experiments' consistency and reliability are influenced by this specific hardware configuration, and variations in hardware might lead to different outcomes.

Figures 6.1 and 6.2 demonstrate that the overhead caused by the production of both the intermediate and extended counterexamples is negligible for a large number of tests. The observed runtime difference for the majority of the remaining tests was as much as 50% relative to the tests that were run without generating counterexamples. Note that this difference occasionally appears to be decreasing, which might occur due to random noise. However, a few individual tests produced an overhead of more than 100%. A more thorough examination revealed that these programs typically incorporated

complex quantified permission structures.

Similar to Silicon, numerous tests demonstrated low overhead, and a sizable portion even ran more quickly when counterexample creation was used, as shown Figures 6.3 and 6.4. However, for a small amount of tests, the runtime was doubled while producing a counterexample. Unfortunately, we were unable to identify the precise Viper language features that caused this unusual overhead.

In addition to the timing analysis, we conducted a qualitative examination of the generated counterexamples. For this purpose, we used test files that went beyond simple Viper language feature testing in order to simulate real-world applications. These included algorithms like "Quickselect" [9] and "Binary Search" [7]. Both backends succeeded in generating the anticipated counterexamples for both, the intermediate and the extended representations, making a more in-depth qualitative analysis unnecessary.

Figure 6.1: Graph illustrates the runtime difference between the new intermediate counterexample representation feature and running the test files without any counterexample generation in Silicon.

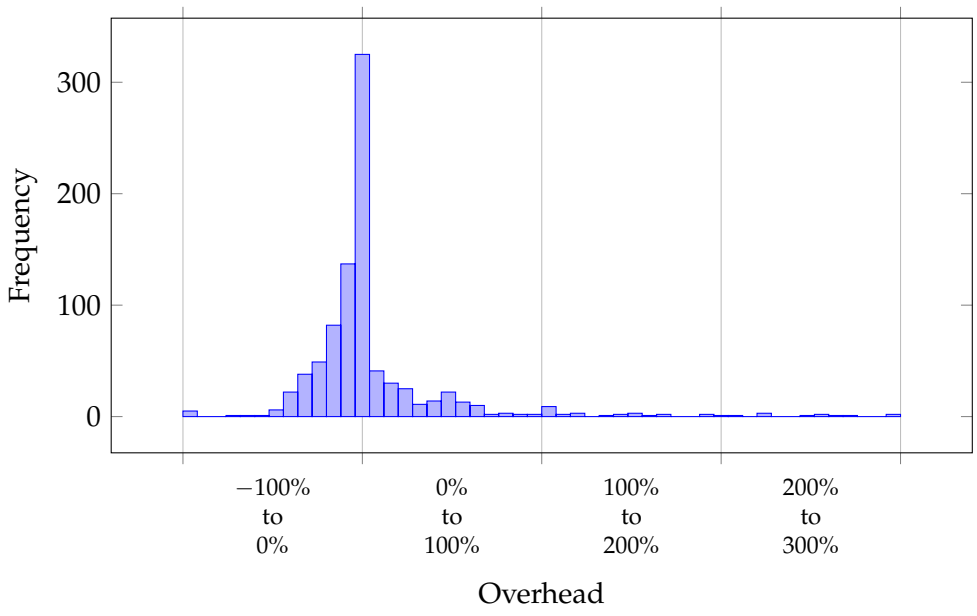


Figure 6.2: Graph illustrates the runtime difference between the new extended counterexample representation feature and running the test files without any counterexample generation in Silicon.

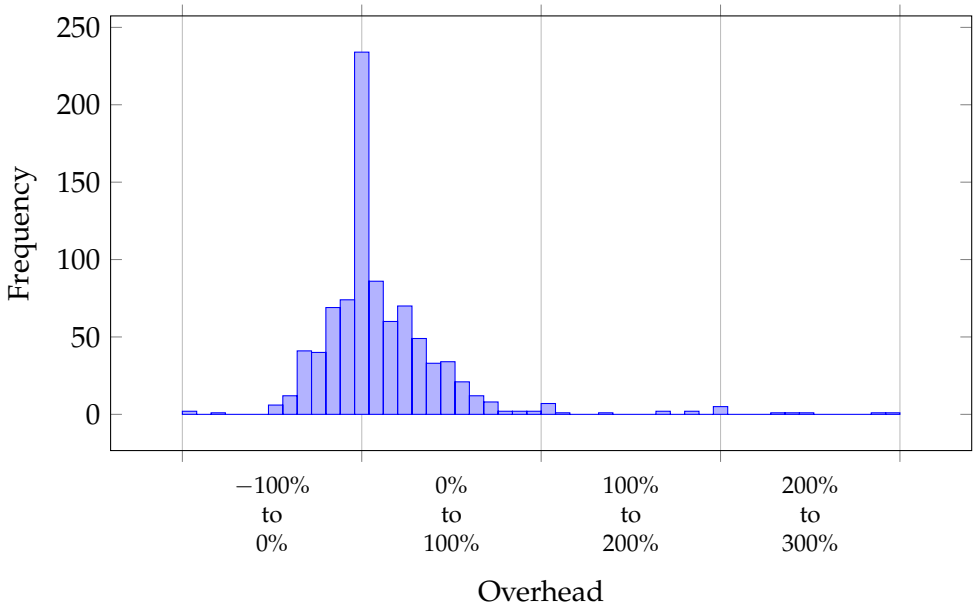


Figure 6.3: Graph illustrates the runtime difference between the new intermediate counterexample representation feature and running the test files without any counterexample generation in Carbon.

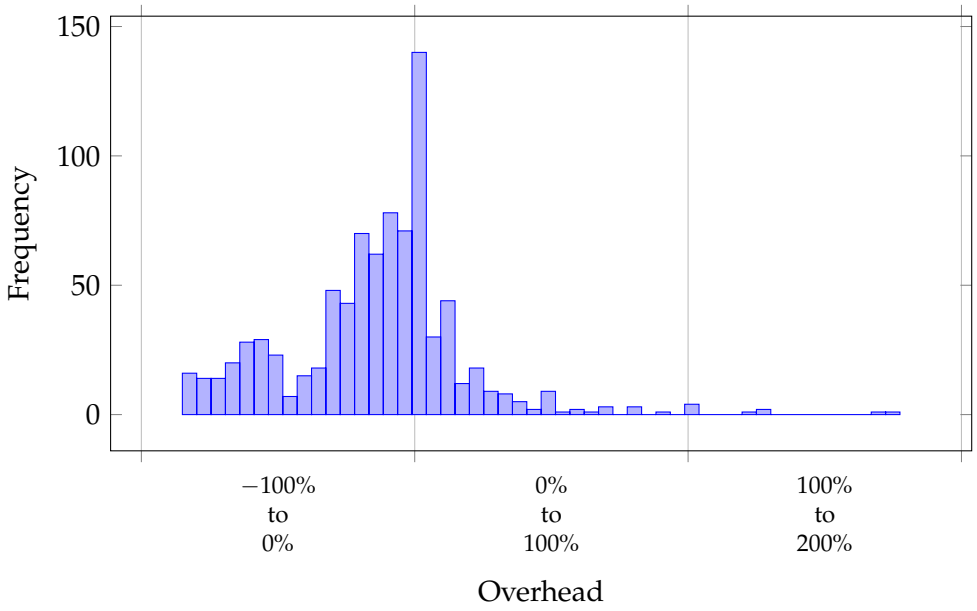
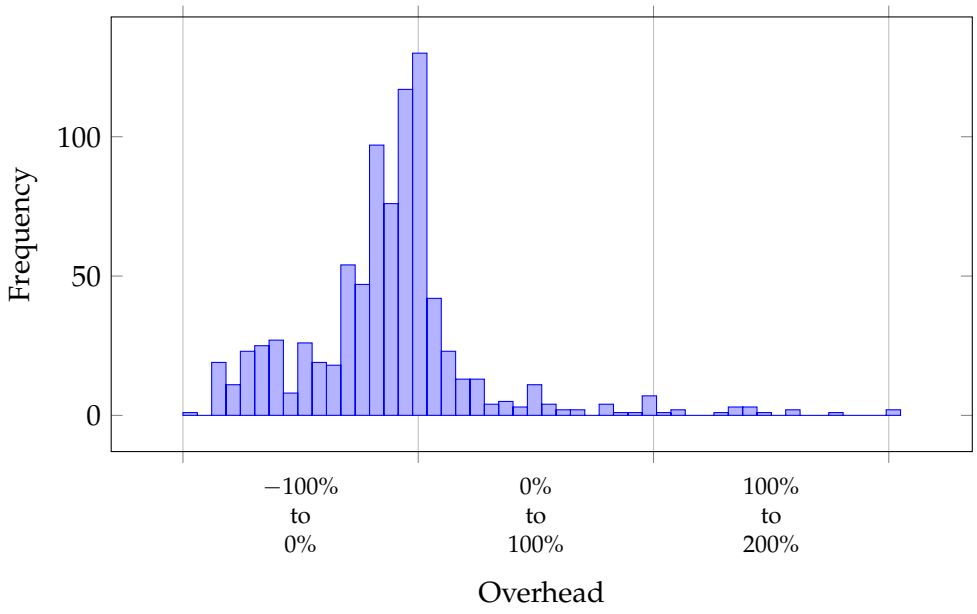


Figure 6.4: Graph illustrates the runtime difference between the new extended counterexample representation feature and running the test files without any counterexample generation in Carbon.



Chapter 7

Conclusion

This thesis aimed to improve the quality and presentation of the generated counterexamples in Viper and make them backend independent. The work can be divided into three main goals:

- Establishing a standard format for counterexamples that could be used across both backends.
- Enhancing Carbon's ability to support counterexamples to match the existing capabilities of Silicon.
- Augmenting the capabilities of both backend's counterexamples by integrating support for additional program structures, such as predicates, magic wands, and quantified permissions.

Our implementation enhances the usability of the Viper intermediate language by providing advanced feedback in case of verification errors. Further, by separating the counterexample into two evaluation stages, we offer the programmer to choose between a intermediate counterexample and a more comprehensible extended version.

Compared to previous Viper counterexample implementations, our approach provides backend independence for counterexample features, a significant shift from the prior dependence on the Silicon backend. Moreover, we have introduced counterexample support for previously missing Viper types. As a result, our implementation improves the counterexample support in future Viper-based front-ends that rely on Viper.

7.1 Future Work

We will now outline possible next steps extending on our work. Some areas to explore include improving the current generation of counterexamples and using counterexamples of the Viper intermediate language to introduce or

enhance counterexamples in other verification languages based on Viper. Some future projects could include:

- Further decrease the sparsity of counterexamples. That is, increase the number of terms that can be evaluated.
- Extend the Viper Visual Studio Code Extension, to display counterexamples in an interactive way based on the newly created common counterexample format without using a command-line operation.
- Adapt other front-ends such as Nagini [4] or Prusti [18] to the new common counterexample format displayed by both backends.

Bibliography

[1] Fabio Aliberti. *Counterexample Generation in Gobra*. Bachelor's thesis, ETH Zurich, 2021.

[2] Till Arnold. *Optimization of a Viper-Based Verifier*. Bachelor's thesis, ETH Zurich, 2021.

[3] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[4] Marco Eilers and Peter Müller. Nagini: a static verifier for Python. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I 30*, pages 596–603. Springer, 2018.

[5] Cedric Hegglin. *Counterexamples for a Rust verifier*. Bachelor's thesis, ETH Zurich, 2021.

[6] Stefan Heule, Ioannis T Kassios, Peter Müller, and Alexander J Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In *ECOOP 2013–Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings 27*, pages 451–476. Springer, 2013.

[7] Florent Hivert, J-C Novelli, and J-Y Thibon. The algebra of binary search trees. *Theoretical Computer Science*, 339(1):129–165, 2005.

[8] K. Rustan M. Leino. This is boogie 2. Working draft; available at http://research.microsoft.com/en-us/um/people/leino/papers.html, 2008.

[9] Hosam M Mahmoud, Reza Modarres, and Robert T Smythe. Analysis of quickselect: An algorithm for order statistics. *RAIRO-Theoretical Informatics and Applications*, 29(4):255–276, 1995.

[10] Alessandro Maissen. *Adding Algebraic Data Types to a Verification Language*. Practical work, ETH Zurich, 2022.

[11] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*, pages 41–62. Springer, 2016.

[12] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

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

[14] Cédric Stoll. *SMT models for verification debugging*. Master's thesis, ETH Zurich, 2019.

[15] Viper Team. Viper tutorial. https://viper.ethz.ch/tutorial/. Accessed: 2023-03-20.

[16] Nicola Widmer. *Sound Automation of Magic Wands in a Symbolic-Execution Verifier*. Bachelor's thesis, ETH Zurich, 2022.

[17] Felix A Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C Pereira, and Peter Müller. Gobra: Modular specification and verification of Go programs. In *International Conference on Computer Aided Verification*, pages 367–379. Springer, 2021.

[18] Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J Summers. Modular specification and verification of closures in Rust. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 5, New York, NY, USA, oct 2021. ACM.