# Advanced Counterexample Generation in Viper

Bachelor's Thesis Description

Raoul van Doren
Supervised by Dr. Marco Eilers and Aurel Bílý
under Prof. Dr. Peter Müller

March 24, 2023

## 1 Introduction

Viper [1] is an automatic verifier for permission-based reasoning developed at ETH Zurich. It checks the validity of pre- and postcondition, assertions, and safety properties, focusing on permission-based reasoning [2]. Frontend verifiers, such as the modular verifier for Python programs Nagini, can build their verification infrastructure on top of Viper by translating their programs to the Viper intermediate language.

Viper has two backends to verify the correctness of its programs, the Symbolic Execution (SE) backend [3], internally called Silicon, and the Verification Condition Generation (VCG) backend [4], internally called Carbon.

Silicon is based on sound symbolic execution. It generates logical constraints for the state of the program at each point in the Viper program. An automated theorem prover, the Z3 SMT solver, is then used to examine whether the specifications are satisfied by the program to verify the correctness of a program.

Carbon does not directly translate the input Viper program to SMT but instead works by translating the program into Boogie, another verification language. Boogie then generates logical expressions for the Viper program. These verification conditions are then checked using the Z3 SMT solver to determine the correctness of the Viper program.

A counterexample [5] is a set of inputs that violates the correctness of a program during program verification. Counterexamples provide essential information to the programmer as they offer feedback regarding the cause of the verification failure. This information helps the programmer to identify and correct errors more easily. Verification can fail for various reasons: unintended behavior, overly strict specifications, or insufficient automation in the verification backend. Without counterexamples, it can be hard to identify the cause of the problem in the code, especially in complex programs. By providing information about the input that caused the verification to fail, counterexamples make it easier for programmers to correct their code.

Both Viper backends can generate counterexamples. However, the two backends currently provide a different format for their counterexample representation. Additionally, not all parts of the Viper language are supported in the counterexample generation of both backends. In the following, we will describe which parts of the language are included in the current state of counterexample support of the backends:

- Basic / primitive types: These include types like integers, Booleans and permission amounts. Permissions in Viper are rational numbers used to specify which heap locations can be accessed by an operation. Both backends support the representation of these types in their counterexamples.

- References: References are built-in types that hold values pointing to objects. The counterexamples of both backends support references.

- Sequences: Finite sequences of elements of a particular type are a built-in type in Viper. Their representation in counterexamples is implemented in Silicon but not in Carbon. However, a problem with the representation of sequences in Silicon's counterexamples is that it does not output exact values for every index and the sequence length. Instead, it interpolates to construct values for all entries based on partial knowledge, since SMT solvers typically provide partial models. This means that the SMT solvers report values for a finite number of argument combinations rather than for all possible argument combinations. Although an exact representation of every value in the sequence might be unfavorable for long sequences, Silicon's interpolation technique uses the available information to provide a complete representation of the sequence.

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

Listing 1.1: The code defines a method "update" that takes a sequence of integers which 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 value of the updated sequence are not equal. As the method sets the second and third value of the updated sequence to 42, the third postcondition is violated.

```
        counterexample:
model at label old:
values <- (Seq<Int>!val!1): [42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42]
updatedValues <- (Seq<Int>!val!4): [0, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42]
on return:
values <- (Seq<Int>!val!1): [42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42]
updatedValues <- (Seq<Int>!val!4): [0, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42]
```

Listing 1.2: The counterexample generated by Silicon for the code from Listing 1.1 shows the "values" and "updateValues" sequences with specific values for the old and return label.

```
FailureContextImpl(Some(values -> T@U!val!6
updatedValues -> T@U!val!20))
```

Listing 1.3: The counterexample generated by Carbon for code from Listing 1.1 only gives names for the "value" and "updatedValues" sequences, but no information about the integer values in the sequence.

- Sets and Multisets: While sets are supported for the counterexamples generated by Silicon, neither backend supports multisets. Due to the partial model received from the SMT solver, Silicon's counterexample representation for sets has the same problem as its representation of sequences.

- Heap: Viper has a built-in notion of a program heap and permissions to memory locations. Accessing a heap location requires the corresponding permission. Recursive predicates, magic wands, and quantified permissions are the features supported by Viper to specify and reason about unbounded heap structures.

  - Silicon maintains a symbolic heap during symbolic execution, and knows, at any point in the program, which fields and predicates are accessible, as well as their receivers and permission amounts. Field chunks map a field receiver to a location value and a permission amount.
  - Carbon, on the other hand, encodes the heap into a global variable, a map from locations to values. Permissions are defined using permission masks stored in the variable Mask, which map locations to Booleans.

  While Carbon does not provide any counterexample support for heaps, Silicon processes field chunks and predicate chunks. However, the final counterexample representation of the symbolic execution backend only covers field chunks.

- Predicates: In Viper, recursive predicates are used to specify unbounded data structures on the heap, such as linked lists and trees. Neither backend shows predicates in its counterexample representations.

```
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 2.1: The code implements a method "setTuple". The postcondition of this method ensures that the sum of the values of "this.left" and "this.right" is one more than the sum of their old values, but this condition does not hold.

```
        counterexample:
model at label old:
this <- Ref ($Ref!val!0) {
}
l <- -410
r <- -410
on return:
this <- Ref ($Ref!val!0) {
}
l <- -410
r <- -410
```

Listing 2.2: The counterexample generated from Silicon for the code from Listing 2.1 shows specific values for the fields, but has no representation of the predicate.

```
FailureContextImpl(Some(r -> (- 886)
this -> T@U!val!7
l -> 264))
```

Listing 2.3: The counterexample generated from Carbon for code from Listing 2.1 shows the values of the local variables. However, it does not show the predicate or directly show the values of the fields inside the predicate.

- Magic Wands: Magic wands are a binary connective that provides the ability to express guarantees about future additions to the state by promising that if combined with a state satisfying one assertion, it can be exchanged for the other assertion. In Viper, magic wands are used to represent permissions to partial data structures. Neither of the two backends support this element in their counterexample representations.

- Quantified Permissions: Quantification permissions are used to specify unbounded heap structures in Viper. Both Carbon and Silicon do not yet provide counterexample support for this feature.

```
field first : Ref
field second : Ref
method inc(nodes: Set[Ref], x: Ref)
  requires forall n:Ref :: { n.first } n in nodes ==>
    acc(n.first) &&
    (n.first != null ==> n.first in nodes)
  requires forall n:Ref :: { n.second } n in nodes ==>
    acc(n.second) &&
    (n.second != null ==> n.second in nodes)
  requires x in nodes
{
  var y : Ref
  if(x.second != null) {
    //permissions covered by preconditions
    y := x.second.first
    //violation of permissions
    y.first := null
  }
}
```

Listing 3.1: The code implements a method called "inc": If the "second" field of the reference "x" is not null, the "first" field of "x" is assigned to "y". The first assignment in the "if" block is allowed as the preconditions ensure that the reference is accessible and that the value is in the "nodes" set. However, the second assignment in the "if" block violates the permissions as permission to "y.first" is not given in the case that node "y" is null. Thus, we have a quantified permission violating correctness.

```
        counterexample:
model at label old:
nodes <- (Set<$Ref>!val!0): {}
x <- Ref ($Ref!val!0) {
}
```

```
y <- Null($Ref!val!2)
on return:
nodes <- (Set<$Ref>!val!0): {}
x <- Ref ($Ref!val!0) {
}
y <- Null($Ref!val!2)
```

Listing 3.2: The counterexample generated from Silicon for the code from Listing 3.1 does not show any values for the set of nodes, for the nodes and for their fields.

```
FailureContextImpl(Some(nodes -> T@U!val!6
x -> T@U!val!11
y -> T@U!val!17))
```

Listing 3.3: The counterexample generated from Carbon for the code from Listing 3.1 only shows names for the nodes, but it assigns no values to the fields of a node which prove the violation of the permissions in the code.

- Domains: Domains are types that are defined by the programmer. They consist of a type name and a block for newly defined function declarations and axioms. Silicon outputs a value for all domain function instances of each domain and type instantiation in its counterexamples [6]. Carbon does not include domains in its counterexample representation.

This thesis aims to make the quality and the representation of the generated counterexample independent of which Viper backend was used. That will require: (1) creating a common format for counterexamples across the two backends, (2) extending the verification condition generation backend's counterexample support to the current counterexample generation capabilities of the symbolic execution backend, and (3) improving the capacities of the counterexamples of the symbolic execution backend by adding support for quantified permissions and possibly other features.

# 2 Goals

## 2.1 Core Goals

- First, create a common counterexample format for both backends, making it easier for the programmer to use. To achieve this standard for-

mat for counterexample representations, the following points should be worked on:

  - Examine the existing counterexample representation of Silicon and determine which elements are specialized to its counterexamples. Those elements should be changed to the newly introduced consistent counterexample format chosen for both backends.
  - Identify the counterexample entries the two backends have in common and consolidate them into one general entry such that both backends can use the standardized counterexample representation.
  - Find a general representation for counterexamples with partial knowledge.

- Add support to the existing counterexample representation of the verification condition generation backend such that it provides the same functionality for counterexamples as the symbolic execution backend. Design a syntax that represents those elements in the same counterexample format as the symbolic execution backend does. This includes analyzing how the symbolic execution backend provides counterexample support for the types and what information (e.g., fields, values, permission amounts) it includes in its counterexamples for those types. The elements which have to be inspected are:

  - Sequences and Sets
  - Domains functions
  - Heap representations

- Add the representation of predicates to the support for heap counterexamples in both verifiers.

- Incorporate predicate snapshots into Silicon's counterexample support, which includes the representation of contained permissions and values extracted from the predicate.

- Add the representation of quantified permissions into Silicon's support for counterexamples. To do so, support for quantified chunks needs to be introduced to the current counterexample generation, which includes extracting information from the quantified chunks and looking up the values of inverse functions and permission expressions.

## 2.2 Extension Goals

- Include the support for magic wands into the counterexample representation generated by both Viper backends. First, explain what values can be extracted from the feedback received from the Z3 and Boogie verifier and the resulting heap changes. Identify the use of the previously derived information and determine which information would be necessary for the counterexample presented to the user. While doing so, magic wands should be treated the same as predicates.
- Improve the functionality of Silicon and Carbon to map the values of heap-dependent functions to specific states in the shown counterexample. This includes developing a sound method for extracting information about heap-dependent functions.
- Adapt Nagini [7] [8] to the new common counterexample format displayed by both backends so that it can be combined with both backends. Additionally, identify which previously implemented features for counterexamples are not included in Nagini (such as quantified permissions and predicate snapshots) and add them, if practical, to Nagini's existing counterexample support.
- Extend the Visual Studio Code Extension, which provides interactive IDE features for Viper, to display counterexamples in the previously discussed format for both backends to the user without using the command-line.

# References


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

[2] 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.

[3] Malte H Schwerhoff. *Advancing automated, permission-based program verification using symbolic execution*. PhD thesis, ETH Zurich, 2016.

[4] 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.

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

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

[7] 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.

[8] Marco Eilers. *Modular Specification and Verification of Security Properties for Mainstream Languages*. PhD thesis, ETH Zurich, 2022.