ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

SMT Models for Verification Debugging

Master Thesis

Cédric Stoll

Mai 19, 2019

Advisors: Dr. Alexander Summers, Arshavir Ter-Garielyan
Department of Computer Science, ETH Zürich

Abstract

Advancements of the SMT solvers led to significant improvements in the field of program verification. As the SMT solvers became faster and faster, the task of verifying software became feasible. Trying to formally verify software code can be a cumbersome task even with the help of highly automated tools, like Viper. A failed verification attempt leaves the developer with only an exception and no further help to identify the error he has made.

In this thesis we used the underlying SMT solver to generate counterexamples for a failing verification attempt. We used an experimental approach to cover a big enough subset of the Viper language to create a prototype that is capable of generating counterexamples for a wide range of Viper programs. These counterexamples are than visualized and could be shown directly in the IDE to further simplify the debugging process. In comparison to previous iterations of counterexample generators, this implementation improves on the theory support and removes the restriction of a bounded search space.

Contents

Contents **iii**

1 Introduction **1**

2 Approach **5**
2.1 Counterexample Generation Process 6
2.2 Experimental Approach 7

3 Counterexample generation **9**
3.1 Basic Features 9
3.1.1 Variables 9
3.1.2 Heap Independent Functions 12
3.1.3 Permissions 14
3.2 Extended Features 15
3.2.1 Quantified Permissions 16
3.2.2 Function Graph 21
3.2.3 Quantified Permission V2 22
3.3 Open Problems 26
3.3.1 Heap dependent Functions 26
3.3.2 Predicates 28
3.3.3 Set, Sequences, Multisets 29
3.3.4 Matching Program States 29

4 Implementation **31**
4.1 Overview 31
4.1.1 Getting started: Generate a Counterexample 31
4.2 Input Preparation 32
4.2.1 Post-processing the SMT2 file 33
4.3 Counterexample Generator 33
4.3.1 Parsing 33

4.3.2 Building the counterexample . . . 34
4.3.3 Detailed explanations . . . 34
4.4 Visualizer . . . 38
4.4.1 Store . . . 38
4.4.2 Functions . . . 38
4.4.3 Heap . . . 40

5 Evaluation **41**
5.1 Example Set . . . 41
5.2 Case Study . . . 41
5.2.1 Arithmetic Example . . . 42
5.2.2 Reference Field Example . . . 42
5.2.3 Increment Function . . . 43
5.2.4 Array Maximum . . . 44
5.3 Comparison to the Project Description . . . 47
5.3.1 Core Goals . . . 47
5.3.2 Extension Goals . . . 48
5.4 Comparison to other Tools . . . 48

6 Conclusion and Future Work **51**
6.1 Future Work . . . 51

A Appendix **53**
A.1 Examples . . . 53

Bibliography **57**

Chapter 1

Introduction

Improvements of the SMT (Satisfiability Modulo Theories) solvers in the last years had led to significant advancements in the field of program verification. As SMT solvers became faster and faster, the task of verifying software automatically became feasible. One example for such a software verification platform is Viper [11].

Trying to formally verify software code can be a cumbersome task even with help of highly automated tools. A failed verification attempt leaves the developer with an "Assertion might not hold" exception and no further help in identifying the possible errors he has made.

A debugger that could provide some hints, e.g. counterexamples to the current proof attempt, would strongly facilitate debugging. Alessio Aurecchia's Master thesis "Visual Debugging for Symbolic Execution" [2] (using Ruben Kälin's work "Advanced Features for an Integrated Verification Environment" [10] and Ivo Colombo's "Debugging Symbolic Execution" [4]) tackled this problem and developed a counterexample generation engine. For a subset of the Viper language it can generate small counterexamples and displays them visually in the IDE. To generate possible counterexamples the software modelling tool Alloy [1] has been used. Using the symbolic execution log, generated by one of the Viper verifiers (Silicon [12]), they generated an Alloy program for a failed assertion. The model for this Alloy program then corresponded to a possible counterexample for the original program. This approach has a few drawbacks: Alloy does not natively support theories and it has to bound the range of integers it considers. This led to approximations, which in consequence resulted in spurious counterexamples.

In this thesis, an alternative approach to generate counterexamples is developed. Silicon uses an SMT solver, Z3 [5][6] to be specific, to verify the code and specifications. If a verification step fails, Z3 is used to produce a counterexample to the current failing assertion. After a model has been

obtained from Z3, it needs to be translated back to a counterexample of the program. The advantage of using Z3 instead of Alloy for generating counterexamples comes in the native theory support that Z3 has. This theory support will allow us to model the theory constraints precisely and we will not get spurious models based on an incomplete theory support.

The following example of a realistic Viper program illustrates how a counterexample supports the process of finding the current problem in the verification task.

```
1  field val : Int
2
3  define access(a) forall j: Int :: 0 <= j && j < len(a) ==>
       acc(loc(a, j).val)
4  define untouched(a) forall j: Int :: 0 <= j && j < len(a)
     ==> loc(a, j).val == old(loc(a, j).val)
5
6  domain Array {
7      function loc(a: Array, i: Int): Ref
8      function len(a: Array) : Int
9      function first(r: Ref) : Array
10     function second(r: Ref) : Int
11
12     axiom injectivity {
13         forall a: Array, i: Int :: {loc(a, i)} first(loc(a
               , i)) == a && second(loc(a, i)) == i
14     }
15
16     axiom length_nonneg {
17         forall a: Array :: len(a) >= 0
18     }
19 }
20
21 method find_max(a:Array) returns (max:Int)
22     requires access(a)
23     // non-negative integers
24     requires forall i:Int :: 0 <= i && i < len(a) ==> loc(
           a, i).val >= 0
25
26     ensures access(a) && untouched(a)
27     ensures len(a) == 0 ==> max == -1
28     ensures forall i:Int :: 0 <= i && i < len(a) ==> loc(a
           , i).val <= max
29 {
30     if(len(a) == 0)
31     {
32         max := -1
33     }
34     else
35     {
```

```
36          var i:Int := 0
37          while(i < len(a))
38              invariant access(a) && untouched(a)
39              invariant i >= 0 && i <= len(a)
40              invariant forall j:Int :: { loc(a, j) } j >= 0
                    && j < i ==> loc(a, j).val <= max
41          {
42              i := i + 1
43              //Conditional statement might fail. There
                    might be insufficient permission to access
                    loc(a, i).val
44              if(loc(a, i).val > max)
45              {
46                  max := loc(a, i).val
47              }
48          }
49      }
50  }
```

The shown example program employs a method to find the maximum inside an array of non-negative integers. Being familiar with the structure of software code, one will easily identify the underlying fault. The index `i` has been increased before the array `a` is been accessed and if it reaches its maximum possible value of `len(a)-1` and its entering the loop body, this will cause an out-of-bounds array access.

A counterexample to this program could consists of some array with length `5` and `i = 5` at the state of the verification failure. This would show the developer that the code tries to access the array outside of its bounds. There are a lot of other examples where a counterexample can be useful and we hope that the one above gave a short introduction to the idea.

This thesis assumes that the reader already has some knowledge about program verification, in particularly about the Viper tool-stack (Silicon and Z3, knowledge about Carbon is not needed in this thesis). The Program Verification course at ETHZ covers all the prerequisites for this thesis.

Chapter 2

Approach

In this chapter we look at the approach we took to create counterexamples for failing verification attempts in Silicon. In every step during the verification, Silicon performs a validity check using the path conditions *Path* and the current proof objective (the next step in the verification *Step*).

$$Path \models Step \quad \equiv \quad Path \Rightarrow Step \quad \equiv \quad \neg Path \vee Step$$

As we only have a satisfiability solver we have to negate the formula.

$$\neg(\neg Path \vee Step) \quad \equiv \quad Path \wedge \neg Step$$

This formula is only satisfiable if the path conditions are satisfiable but the next step is not. Therefore if it is satisfiable there exists a model in which the current verification fails. Only if it is unsatisfiable Silicon will continue, in all other cases (SAT, UNKNOWN and TIMEOUT) it will stop and respond with a verification failure. As counterexamples should be created once a case of failure is occurring, the UNSAT case can be neglected, as in such a case the procedure would have never been started. Additionally we will also ignore the TIMEOUT case.

If a formula is satisfiable and the SMT solver converges to a SAT or UNKNOWN response, the solver can produce a model (see the entry for `(check-sat)` in the SMT-LIB standard [3]). We can then use this model, as it represents a counterexample on the SMT level, and map it back to the original Viper code. This would than be a valid counterexample to the failing verification attempt and the user will be provided with a concrete reason why the code does not verify and not just a simple error message.

Queries that come from Silicon almost always result in a UNKNOWN response, as it heavily depends on universal quantifiers. Previous to this work, we assumed that *partial models* that come from a UNKNOWN response are not useful. Nevertheless Our first task was to investigate the usefulness of these types

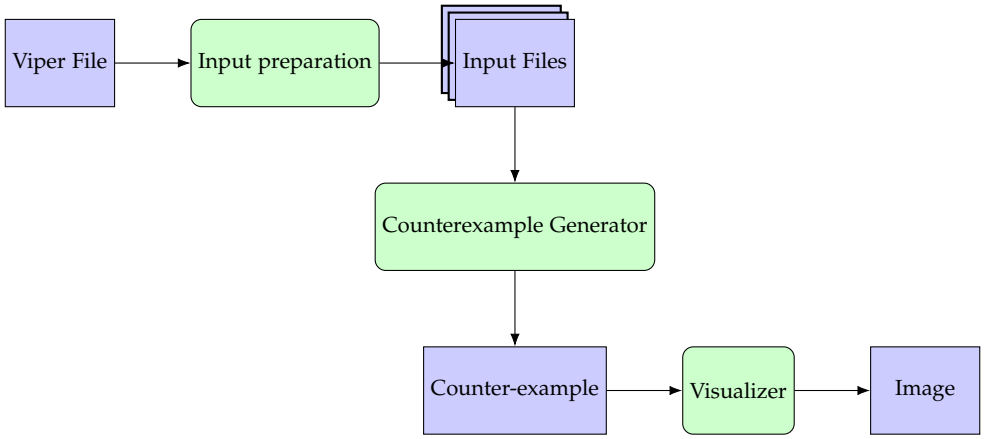

Figure 2.1: Overview of the counterexample generation process

of models. During the first weeks we realized that for all of our test cases the model contained a complete counterexample. Because we could not find a single viper program where this was not true, we made the following assumption for this thesis: Every model, independently of the satisfiability result (SAT or UNKNOWN), contains a valid counterexample to the failing viper program.

2.1 Counterexample Generation Process

We will now go through our newly-developed procedure that generates a counterexample given a Viper source file. An overview over the counterexample generation process for a given Viper source file is shown in Figure 2.1. In short, the failing Viper code undergoes an input preparation step which creates the required input files which are fed into the Counterexample Generator. The created counterexample is passed on to the Visualizer that creates an output in the form of an image. More information about the input preparation is included in this section, the counterexample generation will be treated in Chapter 3 and the implementation of the visualizer is covered in Chapter 4.

Figure 2.2 depicts an outline of the input preparation stage. We generate a SMT2 file[1], a Z3 model for this SMT2 file and a SymbExLog (Symbolic Execution Log)[2] for each failing method in the Viper source. This allows us to independently generate a counterexample for each failing method. It starts with running Silicon on a given Viper file. This will generate a list of all failing methods, a combined SMT2 file and a combined SymbExLog. We

[1] A SMT2 File contains statements which satisfy the SMT-LIB 2.x standard, see the documentation [3].

[2] An introduction to Symbolic Execution can be found at [9].

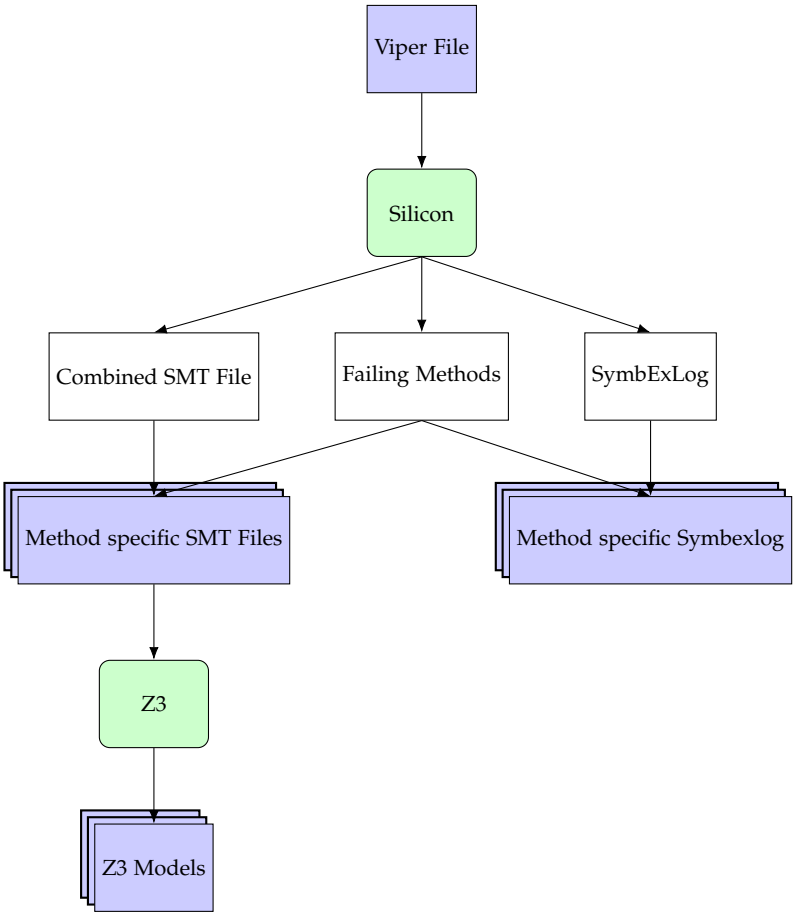

Figure 2.2: Input preparation for the counterexample generation process.

can split the SMT2 file and SymbExLog into one for each method. Executing Z3 for each SMT2 file will get us a partial model for each of those. At the end we have a directory for each failing method, where each directory contains a SMT2 file and the SymbExLog.

2.2 Experimental Approach

At the beginning of this thesis we decided to use an experimental approach to create this process for generating the counterexamples. Therefore we did not look at the transformation work that Silicon does and then tried to define rules over all possible SMT2 files and SymbExLogs. We instead started with a simple Viper example and created a procedure that could handle this type of problem. If we found a solution to the problem class we continued with a more difficult example, an additional Viper feature or a more difficult combination of features. In the end we got a set of rules that handles a large

subset of Viper programs. This type of approach has a disadvantage as we handled Silicon almost like a black box, we will never reach a state where we can say with absolute certainty that we cover all possible cases. This means, we will never have a guarantee for completeness using this approach.

Chapter 3

Counterexample generation

In this chapter we will go through different Viper examples, discuss how the corresponding counterexamples could look like and how it can be generated. Each example will contain progressively more Viper features or will expose problems of older solutions. The goal at the end is to extract general rules for generating a counterexample given some Viper file that contains a combination of Viper features.

Before we can start looking into the first example, we need to specify what a counterexample contains. One possibility is to display the latest values for each store variable, heap location and the function models. Another would be to give the input arguments for the failing method that result in a verification failure. In this thesis we decided to display the state where the verification failure occurs. This state contains the store variables, heap values and function models (both user-defined and internal ones generated by Silicon). Later in the chapter we will also look into features where this definition of a counterexample could not be enough and we will add more information to the counterexample.

3.1 Basic Features

In this introductory section we will discuss the three basic components of every counterexample: store variables, heap values and functions. It is crucial to support these, otherwise working on more complex Viper features would be meaningless.

3.1.1 Variables

Let us start with an easy example: a Viper program with only variables and a control flow statement.

```
1  method test(b:Bool, i: Int)
2  {
3      var x: Int := i
4      x := x + 1
5      x := 2 * x
6
7      if(b)
8      {
9          x := 10
10     }
11
12     //Assert might fail. Assertion x == 10 might not hold.
13     assert x == 10
14 }
```

In this code, the assertion `x == 10` fails if `b = false` and `i != 4`. We will now look at two of the generated input files: the SMT2 statements and the Z3 model. As these files are too big to include in their entirety, we will only show the important parts for the current discussion. Let's start with the variable declarations and the main assertions of the generated SMT2 file[1,2]:

```
1  (declare-const b@0 Bool)
2  (declare-const i@1 Int)
3  (declare-const b@2 Bool)
4  (declare-const i@3 Int)
5  (declare-const x@4 Int)
6  (declare-const x@5 Int)
7  (declare-const x@6 Int)
8
9  (assert (= x@5 (+ i@3 1)))
10 (assert (= x@6 (* 2 x@5)))
11
12 (assert (not b@2))
13 (assert (not (= x@6 10)))
14 (check-sat)
15 ; unknown
```

We are not particularly interested in how Silicon does the transformation from a Viper file to SMT2, we only need to know the SMT2 constants it declares and in which of them the final values for the Viper variables are stored. Even in this small example some important observations can be made. The first one is that Silicon keeps the variable names that were used in the Viper file and the second one is that the code has been transformed into a single assignment form. We can observe that the constants contain an ascending identifier. We could assume that for each Viper variable the

[1] For more informations about SMT2 statements, look at the SMT-LIB 2.6 standard [3].

[2] In this thesis we simplified all identifiers generated by Silicon. Normally they would look like `b@0`, we simplified them to `b@0` as we do not use the second numerical value.

corresponding constant with the biggest identifier is the last version before the failing assertion. But to be sure we can use additional information from the SymbExLog. If we look at the last entry before the verification failure, we can find the following information about the store:

```
1 store:
2 [
3     {value: b -> b@2, type: Bool},
4     {value: i -> i@3, type: Int},
5     {value: x -> x@6, type: Int}
6 ]
```

Here we see for each variable in the Viper file, an entry what name the corresponding constant has and what type it is. For this example our assumption that the biggest identifier would correspond to the last version would have been correct. Now we will look at the Z3 model.

```
1 b@2 -> false
2 x@5 -> 6
3 i@3 -> 5
4 x@6 -> 12
```

This was the whole model that Z3 generated for this example, we can see that it did not assign every constant from the SMT2 query a value[3]. But we can see that the counterexample consist of the following store values `b = false`, `x = 12` and `i = 5`.

As a solution for this example we would then generate the following counterexample.

```
1 Store: {
2     b = false,
3     x = 12,
4     i = 5
5 }
```

Rules

From this example we can propose the following rules:

- Every constant that corresponds to a Viper variable or parameter needs to be part of the store in the counterexample.
  - Extension 1: Remove constants that do not have a value in the model.

[3] We could enforce a complete model with the Z3 parameter `"model.completion=true"`. In the current version we enforce this, as we want to make sure that every user-defined variable has an assignment.

– Extension 2: Over all the constants that represent the same variable, keep only the constant with the highest identifier.

3.1.2 Heap Independent Functions

The next feature we want to include in our supported subset of Viper is functions. As most of the built in features of Viper directly translate to some functions on the SMT level we need support for it. As for variables, lets take a look at a basic example that uses a function. There are cases (like the example below) where the function is not a important part of the counterexample because the implementation is given by the user and therefore already known. But the model for the SMT2 query will look the same whether the implementation is given or not. Therefore, we can use either of these cases to come up with the rules to handle functions.

```
1  function inc(x:Int): Int
2      ensures result == x + 1
3  {
4      x + 1
5  }
6
7  method test(x:Int)
8  {
9      //Assert might fail. Assertion inc(x) == 2 might not
           hold.
10     assert inc(x) == 2
11 }
```

In this code, we have an increment function, an assertion with `inc(x) == 2` and no restrictions on `x`. Therefore this assertion fails for every `x != 1`. But again, lets take a look at the generated SMT2 code:

```
1  (declare-fun inc ($Snap Int) Int)
2
3  (assert (forall ((s@$ $Snap) (x@0 Int)) (!
4      (= (inc s@$ x@0) (+ x@0 1)))
5      :pattern ((inc s@$ x@0))
6    )))
7
8  (declare-const x@0 Int)
9  (declare-const x@1 Int)
10
11 (assert (not (= (inc $Snap.unit x@1) 2)))
12 (check-sat)
13 ; unknown
```

We can see that the `inc` function got an additional `$Snap` parameter. This *snapshot* represents heap information the function depends on. In our case

the function is heap independent, therefore all function cases will have as an argument the empty snapshot `$Snap.unit`.

```
1 x@1 -> 237
2 inc -> {
3     $Snap.unit 237 -> 238
4     else -> #unspecified
5 }
```

In the Z3 model we see that it chose to set `x = 237`; this is not equal to 1 and therefore a valid counterexample. Also we can observe the model for the `inc` function. It has a case for `inc(237)` with the correct result of `238`. Every other case is unspecified[4]. This does work for the specific case of `x = 237` but does not satisfy the postcondition of the `inc` function (`result == x + 1`). This is the default behaviour of Z3, as the postcondition of the function gets modelled as a universal quantifier and quantifiers only apply if they are triggered. Therefore we see the implications of quantifiers only from cases that are present in the example. An implication of this is, that the generated models (in almost every example that contains a function) will not satisfy the pre- or postconditions of the function for all possible input values. But as we are only interested in one specific counterexample and not about all the possible states, this incompleteness is not a problem for us and we can ignore it.

For this example the following counterexample would be a valid solution.

```
1 Store: {
2     x = 237
3 }
4 Functions: {
5     inc = {
6         $Snap.unit 237 -> 238
7     }
8 }
```

Rules

For functions, we can now propose the following rules to incorporate a representation for them into the counterexample.

- Take every function model that corresponds to a user-defined Viper function and add the function definition to the counterexample
  - Open Problem 1: This will include the snapshot argument into the counterexample, even though the user does not know about this parameter.

[4] We achieve this behaviour with setting the Z3 parameter `model.partial = true`.

– Open Problem 2: The mapping from the function calls inside the code to the function cases could in some cases be difficult for the user. For example if the parameters are not of type integer it can get harder. We need some visual approach of mapping a function call to a function case or directly the result of this application. For a difficult example look at `inc` function in the counterexample in Section 3.3.1.

3.1.3 Permissions

The next feature we want to support are permissions. Permissions are the Viper feature to talk about heap memory locations. Again, we will look into a simple example.

```
1  field val : Int
2
3  method access(r:Ref)
4      requires acc(r.val, write)
5      ensures acc(r.val, write)
6  {
7      r.val := r.val + 1
8      //Assert might fail. Assertion r.val == 1 might not
           hold.
9      assert r.val == 1
10 }
```

The example above will fail if `r.val` initially does not have a value of `0`; the simplified SMT2 code looks like this:

```
1  (declare-const r@0 $Ref)
2  (declare-const r@1 $Ref)
3  (declare-const $t@2 Int)
4  (declare-const $t@3 Int)
5  (declare-const val@4 Int)
6
7  (assert (not (= r@1 $Ref.null)))
8  (assert (= val@4 (+ $t@2 1)))
9
10 (assert (not (= val@4 1)))
11 (check-sat)
12 ; unknown
```

We can see the constants defined for the reference parameter `r` (`r@0` and `r@1`) and some assertions over the integer constants that will represent the field values of this reference (`$t@2`, `$t@3` and `val@4`). But inside the SMT2 code we do not see an assertion or some other way of linking these two groups of constants together. To get the information that `val@4` is the field `val` of the reference `r@1` we need to take a look at the SymbExLog. There we find the following JSON entry:

```
1 heap:[r@1.val -> val@4 # W]
```

This entry says that the reference `r@1` has a field `val` and this field has the value of the constant `val@4`. After the `#` comes the permission amount, in this example we have `W` (write, full) permission on this field. Finally we can take a look at the given model for the SMT2 query.

```
1 val@4 -> 720
2 r@1 -> $Ref!val!0
3 $t@2 -> 719
4 $Ref.null -> $Ref!val!1
```

Combining this Z3 model with the additional information from the SymbExLog we now know that `val@4` represents the final `r.val` with the value `720`. Using the rules described in the variables section we already have the store part of this counterexample covered. For the heap section we will now include one entry for each permission in the SymbExLog heap list. In our case this would add an entry for `r.val` with the value `720` (the value Z3 assigned to `val@4`). Combining this would then lead to the following counterexample:

```
1 Store: {
2     r = $Ref!val!0
3 }
4 Functions: {}
5 Heap: {
6     r.val = 720
7 }
```

We could even argue, that the store variable `r` is no longer needed if it has a heap entry for `r.val`. We decided the keep it inside the counterexample, later during the visualisation procedure it can always be hidden if so desired.

Rules

The rules for field permissions look like this:

- For each entry in the SymbExLog heap add an entry to the counterexample heap. In the SymbExLog, the entry has the following structure: `ref.field -> constant # permission`. For the counterexample entry keep the `ref` and `field` values but exchange the `constant` with the assigned value from the Z3 model.

3.2 Extended Features

With these three features covered in the previous chapter we have everything to support basic Viper programs. We support all kinds of store variables,

user-defined functions (or functions generated by Silicon) and basic heap permissions (no quantified permissions or predicates). From now on all new features will build upon these three basic components.

3.2.1 Quantified Permissions

We will use an array example to talk about quantified permissions. As Viper has no native support for arrays, we have to use a domain and specify what an array is ourselves. This modelling approach used four functions in combination with two axioms and quantified permissions to describe the main features of an array. Therefore we can observe the behaviour for both user-defined domains and quantified permissions with this example. As user-defined domains are translated to functions and universal quantifiers, they should already be handled by our basic feature support. But quantified permissions are not just new functions on the SMT level, for this feature we will have to include some new logic to build the counterexample.

```
1  field val : Int
2
3  define access(a) forall j: Int :: 0 <= j && j < len(a) ==>
       acc(loc(a, j).val)
4  define untouched(a) forall j: Int :: 0 <= j && j < len(a)
     ==> loc(a, j).val == old(loc(a, j).val)
5
6  domain Array {
7      function loc(a: Array, i: Int): Ref
8      function len(a: Array) : Int
9      function first(r: Ref) : Array
10     function second(r: Ref) : Int
11
12     axiom injectivity {
13         forall a: Array, i: Int :: {loc(a, i)}
14             first(loc(a, i)) == a &&
15             second(loc(a, i)) == i
16     }
17
18     axiom length_nonneg {
19         forall a: Array :: len(a) >= 0
20     }
21 }
22
23 method test(a:Array) returns (max:Int)
24     requires access(a)
25     requires len(a) > 1
26     requires forall i:Int ::
27         0 <= i && i < len(a) ==> loc(a, i).val >= 0
28 {
29     //Assert might fail. Assertion loc(a, 0).val ==
```

```
30      //  loc(a, 1).val might not hold.
31      assert loc(a, 0).val == loc(a, 1).val
32  }
```

A possible counterexample for the method `test` is the array `[1, 2]`. We will again take a look at the generated SMT2 code. This SMT2 code is heavily simplified. It only covers the declarations of the functions we use to access the values for the array. It does not contain a lot of helper functions, forall assertions or other required components to model the Viper code to SMT2. If you are interested in how this transformation from Viper to SMT2 works, please look at the PhD Thesis from Malte Schwerhoff [12].

```
1  (declare-sort Array)
2  (declare-sort $FVF<Int>)
3
4  ; the 4 user-defined domain functions
5  (declare-fun second ($Ref) Int)
6  (declare-fun first ($Ref) Array)
7  (declare-fun len (Array) Int)
8  (declare-fun loc (Array Int) $Ref)
9
10 (declare-fun $FVF.lookup_val ($FVF<Int> $Ref) Int)
11
12 (declare-const a@2 Array)
13 (declare-const sm@10 $FVF<Int>)
14 (declare-const $t@5 $FVF<Int>)
15
16 (assert (> (len a@2) 1))
17
18 (assert (not (=
19     ($FVF.lookup_val sm@10 (loc a@2 0))
20     ($FVF.lookup_val sm@10 (loc a@2 1)))))
21
22 (check-sat)
23 (get-model)
24 ; unknown
```

On the Viper level, an array access has been modelled as `loc(a, 0).val`. In the SMT2 code above, we can see that this access to the `val` field is not just a constant, like for normal permissions, it uses a function `$FVF.lookup_val`. This function takes a `$FVF<Int>` and a `$Ref` as arguments and returns the `Int` value of the `val` field. Here the sort `$FVF<Int>` represents the current quantified heap chunk. This is the general transformation Silicon does for quantified permissions. Like for normal permissions, the direct link between the `Array` constant `a@2` and its values is not visible in the SMT2 code. For this connection we have to look into the SymbExLog:

```
heap: [
    QA r :: r.val -> $t@5
    # (0 <= inv@9(r) && 0 <= inv@9(r) ==>
        inv@9(r) < len(a@2) ? W : Z)
]
```

Here we can see that for every reference that has a corresponding index (`inv@9` is a function that returns the index for a given reference in the specific quantified heap chunk) between 0 and `(len a@2)` we have `write` permission in the state `$t@5` (`$t@5` is of type `$FVF<Int>`). If we now look at the Z3 model we can complete our counterexample.

```
; Constants
$t@5 -> $FVF<Int>!val!1
sm@10 -> $FVF<Int>!val!0
a@2 -> Array!val!0

; Functions
second -> {
    $Ref!val!0 -> 0
    $Ref!val!1 -> 1
    else -> #unspecified
}
$FVF.lookup_val -> {
    $FVF<Int>!val!0 $Ref!val!0 -> 2998
    $FVF<Int>!val!0 $Ref!val!1 -> 2997
    $FVF<Int>!val!1 $Ref!val!0 -> 2998
    $FVF<Int>!val!1 $Ref!val!1 -> 2997
    else -> #unspecified
}
loc -> {
    Array!val!0 0 -> $Ref!val!0
    Array!val!0 1 -> $Ref!val!1
    else -> #unspecified
}
len -> {
    Array_!val!0 -> 2
    else -> #unspecified
}
first -> {
    $Ref!val!0 -> Array!val!0
    $Ref!val!1 -> Array!val!0
    else -> #unspecified
}
inv@9 -> {
    $Ref!val!0 -> 0
    $Ref!val!1 -> 1
    else -> #unspecified
}
```

As described in Section 3.1.2 about heap independent functions, we will include all user-defined functions in the counterexample. Therefore `first`, `second`, `loc` and `len` will be in the counterexample. Also the variables section of the counterexample is currently correctly handled by the standard rules for variables. They would include the `Array` variable `a@2` with its value, given by the Z3 Model, `Array!val!0`[5]. The only thing that would currently be missing is the heap information, as our previous rules for heap entries only covered field permissions and not quantified permissions.

```
1  Store: {
2      a@2 = Array!val!0
3  }
4  Functions: {
5      first = {
6          $Ref!val!0 -> Array!val!0
7          $Ref!val!1 -> Array!val!0
8      }
9      second = {
10         $Ref!val!0 -> 0
11         $Ref!val!1 -> 1
12     }
13     loc = {
14         Array!val!0 0 -> $Ref!val!0
15         Array!val!0 1 -> $Ref!val!1
16     }
17     len = {
18         Array!val!0 -> 2
19     }
20 }
21 Heap: {
22 }
```

Now we want to include the values that where modelled with the quantified permission in the counterexample. The SymbExLog entry gives us the constant `$t@5`, this constant contains the heap chunk for the failing state and the Z3 Model has the value `$FVF<Int>!val!1` for it. We now have to filter the function `$FVF.lookup_val` with this value for the heap chunk as we are only interested in the values for the final program state. We achieve this filtering with a partial function application, where we used the value `$FVF<Int>!val!1` for the first parameter. With the partial application we get the following function:

```
1  $FVF.lookup_val($FVF<Int>!val!1) = {
2      $Ref!val!0 -> 2998
3      $Ref!val!1 -> 2997
4      else -> #unspecified
5  }
```

[5] Z3 uses this naming scheme for generated values: `SORT!val!ID`

As this function now represents the mapping in the final program state (from heap entries to the `Int` values) we add it to the heap of the counterexample.

```
1 Heap: {
2     $FVF.lookup_val($FVF<Int>!val!1) = {
3         $Ref!val!0 -> 2998
4         $Ref!val!1 -> 2997
5     }
6 }
```

Let us call this partially applied function `fp`. To reconstruct the array from the counterexample the user would now have to go through the following steps:

- Get the value from `a@2` from the counterexample: `a@2 = Array!val!0`
- Partially apply the `loc` function with `Array!val!0` for the `Array` parameter. This will result in the following function that maps `Int` to `$Ref`:

```
1 loc(Array!val!0) = {
2     0 -> $Ref!val!0
3     1 -> $Ref!val!1
4     else -> #unspecified
5 }
```

- With the function `fp` the user has a mapping from the `loc` references to the array integer values. Composing these two `fp(loc(i))` where `i` is some index with `i >= 0 && i < (len Array!val!0)`, would give the following `lookup` function that maps indices to array values:

```
1 lookup = {
2          0 -> 2998
3          1 -> 2997
4          else -> #unspecified
5 }
```

- The reconstructed array of this counterexamples is then [2998, 2997].

Rules

A quantified entry has the following form:

```
QA r ::  r.FIELD -> VAL # PERM.
```

For each quantified heap entry we see in the SymbExLog we perform the following actions:

1. Extract the `FIELD` and `VAL` variable out of the SymbExLog entry.

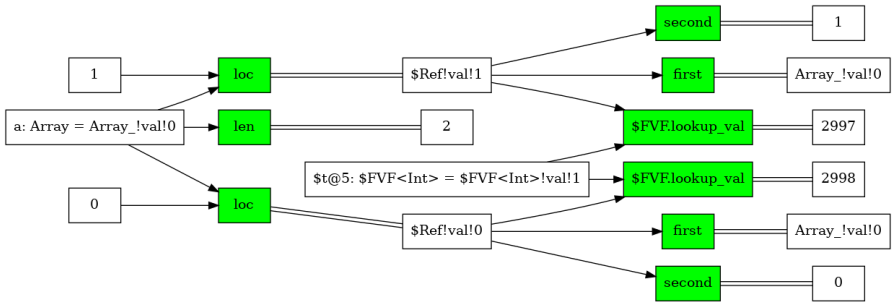

Figure 3.1: Visualisation of a subset of the function graph from the quantified permission example. Where in green we have the functions and in white the argument values.

2. Get the assignments from the Z3 model for the `$FVF.lookup_FIELD` function and the `VAL` variable.
3. Partially apply the function with the model value of the `VAR` variable.
4. Add the partially applied function to the heap of the counterexample.

3.2.2 Function Graph

In the last example we saw that the reconstruction of the counterexample was not an easy task for the end user. We cannot automate this (without any additional information), as the array was modelled by the user and he must do this reconstruction of the original data structure on his own. But only because we cannot do it automatically does not mean we cannot support him.

One way of giving additional support is visualising how the different functions are connected to each other. Our idea was to build a directed acyclic graph for all the function applications (from now on called *function graph* or *function application graph*). The graph should contain all the function applications as nodes that are inside the viper file and also contain all the information Z3 gathered from the different universal quantifiers about these functions. Concrete this means that we have a node for each function case that appeared during the verification. Also we will have a node for each argument of a function and an edge that connect these arguments with the functions, see Figure 3.1 for an example.

The best way to get all this information is directly from Z3, because internally it has this data in the E-Graph and we can be sure that we get all the data Z3 has during the verification. Currently we talk to Z3 only over the SMT2 text interface and we found no way to access these internal data. Therefore we had to reconstruct the required information on our side. This has the disadvantage that we cannot be sure to ever get the exact same

amount of data than Z3 had internally (only if we would reproduce the exact steps that Z3 did).

Construction

The initial set of function calls we get from the SMT2 file. All the additional information about the functions come from quantifiers that get triggered and lead to additional information. Therefore we have to load all the universal quantifier from the SMT2 file and search for combinations of functions that trigger a quantifier. Every triggered quantifier could lead to additional function applications and in consequence to new triggered quantifiers. This could loop endlessly, called a *matching loop*, this means that the implementation does need a mechanism to prevent this from happening. For our implementation see Section 4.3.3. As you can see in the example every function application has a result node. Therefore we have to ask Z3 for all the results of the function calls. After we have gathered all the information we need, we have to create the graph. Here are the steps to create such a graph:

- Create a node for every function application.
- Create a node for every result and connect it to the function application.
- Create a node (if not already present) for every argument of a function application and connect it.

3.2.3 Quantified Permission V2

We will now take a second look at quantified permissions, as there are cases where the previous rules do not get us to the correct result. Here is an example with two integer arrays of length 1:

```
1  field val : Int
2
3  define access(a) forall j: Int :: { loc(a, j) } 0 <= j &&
       j < len(a) ==> acc(loc(a, j).val)
4
5  domain Array {
6      function loc(a: Array, i: Int): Ref
7      function len(a: Array) : Int
8      function first(r: Ref) : Array
9      function second(r: Ref) : Int
10
11     axiom injectivity {
12         forall a: Array, i: Int :: {loc(a, i)}
13             first(loc(a, i)) == a &&
14             second(loc(a, i)) == i
15     }
```

```
16
17      axiom length_nonneg {
18          forall a: Array :: len(a) >= 0
19      }
20  }
21
22  method test(a:Array, a2: Array)
23      requires access(a) && access(a2)
24      requires len(a) == 1 && len(a2) == 1
25  {
26      //Assert might fail. Assertion loc(a, 0).val == loc(a2
            , 0).val might not hold.
27      assert loc(a, 0).val == loc(a2, 0).val
28  }
```

The only difference to the previous quantified permission example is that we compare two elements of different arrays and before we compared two elements of the same array.

Here is the simplified SMT2 code, in this simplified version it is almost exactly the same as the previous example.

```
1  (declare-sort Array)
2  (declare-sort $FVF<Int>)
3
4  ; user-defined functions
5  (declare-fun loc (Array Int) $Ref)
6  (declare-fun len (Array) Int)
7  (declare-fun first ($Ref) Array)
8  (declare-fun second ($Ref) Int)
9
10 (declare-fun $FVF.lookup_val ($FVF<Int> $Ref) Int)
11
12 (declare-const a@0 Array)
13 (declare-const a2@1 Array)
14 (declare-const a@2 Array)
15 (declare-const a2@3 Array)
16 (declare-const sm@18 $FVF<Int>)
17
18 (assert (= (len a@2) 1))
19 (assert (= (len a2@3) 1))
20
21 (assert (not (=
22     ($FVF.lookup_val (as sm@18 $FVF<Int>) (loc a@2 0))
23     ($FVF.lookup_val (as sm@18 $FVF<Int>) (loc a2@3 0)))))
24 (check-sat)
25 ; unknown
```

In the SymbExLog we see now two quantified permissions.

```
heap:[
    QA r :: r.val -> $t@11
        # (0 <= inv@15(r) && 0 <= inv@15(r) ==>
            inv@15(r) < len(a2@3) ? W : Z),
    QA r :: r.val -> $t@5
        # (0 <= inv@9(r) && 0 <= inv@9(r) ==>
            inv@9(r) < len(a@2) ? W : Z)
]
```

And for completeness, here also the Z3 output:

```
; Variables
a2@3 -> Array!val!1
a@2 -> Array!val!0
$t@5 -> $FVF<Int>!val!2
$t@11 -> $FVF<Int>!val!3

; Functions
$FVF.lookup_val -> {
    $FVF<Int>!val!1 $Ref!val!1 -> 3
    $FVF<Int>!val!1 $Ref!val!2 -> 4
    $FVF<Int>!val!3 $Ref!val!1 -> 2
    $FVF<Int>!val!2 $Ref!val!1 -> 3
    $FVF<Int>!val!3 $Ref!val!2 -> 4
    $FVF<Int>!val!2 $Ref!val!2 -> 5
    $FVF<Int>!val!0 $Ref!val!1 -> 3
    $FVF<Int>!val!4 $Ref!val!1 -> 3
    $FVF<Int>!val!4 $Ref!val!2 -> 4
    $FVF<Int>!val!5 $Ref!val!1 -> 3
    $FVF<Int>!val!5 $Ref!val!2 -> 4
    else -> #unspecified
}
inv@9 -> {
    $Ref!val!2 -> (- 8366)
    else -> #unspecified
}
inv@15 -> {
    $Ref!val!1 -> (- 1)
    else -> #unspecified
}
first<Array> -> {
    $Ref!val!2 -> Array!val!1
    else -> #unspecified
}
len -> {
    Array!val!0 -> 1
    Array!val!1 -> 1
    else -> #unspecified
}
loc -> {
```

```
40      Array!val!0 0 -> $Ref!val!1
41      Array!val!1 0 -> $Ref!val!2
42      else -> #unspecified
43  }
44  second -> {
45      #unspecified
46  }
```

If we now apply our previous rules for quantified permissions, we get the following two partial functions:

```
1   Heap: {
2       $FVF.lookup_val($FVF<Int>!val!2) = {
3           $Ref!val!1 -> 3
4           $Ref!val!2 -> 5
5       }
6       $FVF.lookup_val($FVF<Int>!val!3) = {
7           $Ref!val!1 -> 2
8           $Ref!val!2 -> 4
9       }
10  }
```

We can see that both partial function contain two values, even though that the length of both arrays is fixed at 1. The partial functions therefore do not correctly represent the arrays and contain spurious elements. Another problem that we have in this representation, is that it is impossible to match the partial functions to the `Array` variables without looking at the SymbExLog entries.

Currently, we do not have a automated solution for the second problem, matching `Array` variables to the partial functions. Manually this can be done, if the user looks at the SymbExLog permission expression and there he can see the connection to the array (`(len a@2)`). But we have a potential automated solution for the first problem. We only did tests on these types of quantified permissions (the array representation), so there is still work left to do.

Our goal was to remove the spurious elements from the partial functions. We can observe that the `loc` function has the correct entries, one case for each array and it returns the correct reference. But we cannot use the `loc` directly, as there is no obvious connection from the quantified permissions to the `loc` function that is known without user knowledge about how he modelled the array. But what we can use is the permission expression in the SymbExLog. Here is the permission expression for `Array a`:

```
0 <= inv@9(r) && 0 <= inv@9(r) ==> inv@9(r) < len(a@2) ?  W : Z
```

We can observe that the left side has a duplicated expression, let us simplify it.

```
0 <= inv@9(r) ==> inv@9(r) < len(a@2) ?  W : Z
```

We see that we have write permission on a reference if this implication is true. Additionally in the model that we got from Z3, we know that this implication has been enforced. Therefore if the left side is true, we know that we have a positive index (`inv@9` maps references to integers) and, because it is a valid implication, that the index is smaller than `(len a@2)`. But the implication can also be true if the left side is false. In this case we do not learn the right side of the implication but we still have write permission. This is the case that happens in our example above (see the models for both functions `inv@9` and `inv@15` in the Z3 model).

Our solution was then for quantified permission with this form of a permission expression (an implication) to check if the left side is true for each case of the partial function. If the left side is not true for a case, we will remove it from the partial function. If we apply this technique to the current example we would get the following partial functions:

```
1 Heap: {
2     $FVF.lookup_val($FVF<Int>!val!2) = {
3         $Ref!val!1 -> 3
4     }
5     $FVF.lookup_val($FVF<Int>!val!3) = {
6         $Ref!val!2 -> 4
7     }
8 }
```

3.3 Open Problems

For the following sections we only have basic support. This means that the above rules get applies and it does display the variables, function model and heap entries (both normal permissions and quantified permissions). But an improved version should include additions mechanisms to handle them.

3.3.1 Heap dependent Functions

Previously we only looked at heap independent functions, but heap dependent ones do utilize the snapshot feature. Here is a small example that shows what the current solution would do with a heap dependent function. Again we start with the Viper file, it is a small increment function but this time heap dependent.

```
1  field a : Bool
2  field b : Bool
3
4  function inc(config:Ref, x:Int): Int
5      requires acc(config.a, 1/2) && acc(config.b, 1/2)
6      ensures result == (config.a || !config.b ? x + 1 : x)
7  {
8      config.a || !config.b ? x + 1 : x
9  }
10
11 method access(r:Ref, r2:Ref)
12     requires acc(r.a) && acc(r.b)
13     requires acc(r2.a) && acc(r2.b)
14     requires r.a == true && r.b == true
15 {
16     assert inc(r, 5) == 6
17     assert inc(r, 7) == 8
18     r.a := false
19     assert inc(r, 5) == 5
20     assert inc(r, 7) == 7
21
22     assert inc(r2, 7) == 8
23 }
```

A counterexample for this Viper code would be `r2.a = false` and `r2.b = true`. With this configuration the increment function would return 7. Our current rules would generate the following counterexample:

```
1  Store: {
2      r@2 -> $Ref!val!0
3      r2@3 -> $Ref!val!2
4  }
5  Functions: {
6      inc -> {
7          ($Snap.combine $Snap.unit $Snap.unit) $Ref!val!0 5
               -> 6
8          ($Snap.combine $Snap.unit $Snap.unit) $Ref!val!0 7
               -> 8
9          ($Snap.combine ($Snap.combine $Snap.unit $Snap.
              unit) $Snap.unit) $Ref!val!0 5 -> 5
10         ($Snap.combine ($Snap.combine $Snap.unit $Snap.
              unit) $Snap.unit) $Ref!val!0 7 -> 7
11         ($Snap.combine ($Snap.combine $Snap.unit $Snap.
              unit) $Snap.unit) $Ref!val!2 7 -> 7
12         else -> #unspecified
13     }
14 }
15 Heap: {
16     r.a = false
17     r.b = true
```

```
18      r2.a = false
19      r2.b = true
20  }
```

This generated counterexample is correct, but the problem is the representation of the function. As snapshots are Silicon internal a normal end user of Viper does not know what snapshots are. And even if the user knows what snapshots are, a mapping from the snapshot binary trees (these `$Snap.combine` functions build a binary tree) to a tuple of values would be user friendly and appreciated.

3.3.2 Predicates

The same as for heap dependent functions is also true for predicates. The software can generate correct counterexamples but it does not collect and combine every bit of information it could. Here we have a really simple example that contains a predicate.

```
1  field val : Int
2
3  predicate pred(this: Ref)
4  {
5      acc(this.val)
6  }
7
8  method access(r:Ref, x: Int)
9      requires pred(r)
10     ensures pred(r)
11 {
12     unfold pred(r)
13     r.val := x
14     fold pred(r)
15
16     assert x == 2
17 }
```

The predicate does not take an important part of the counterexample, but we want just demonstrate what happens if a counterexample does contain a predicate instance. Predicate instances are part of the heap information in the SymbExLog, in the current example we would find the following entry:

```
pred(x@3; r@2) # W
```

The current rules we have for heap entries only cover normal permissions and quantified permission. Predicates would need an additional set of rules but because if time constraints we just added the predicate instance as it is to the counterexample. Further work is needed to correctly parse this heap

entry and connect it to the already existing parts of the counterexample. The current version would output the following counterexample:

```
1 Store: {
2     r -> $Ref!val!0
3     x -> 0
4 }
5 Functions: {}
6 Heap: {
7     pred(x@3; r@2)
8 }
```

3.3.3 Set, Sequences, Multisets

Viper has built-in support for three data structures set, sequences and multisets. This means that Silicon has built in functions that encode these three. The goal would be that the counterexample would only contain constructs that the user knows from the Viper level. But currently the counterexample would contain these Silicon internal functions. This is correct, but again like in the previous two sections, not the expected output. Future versions should include a mapping back from the counterexample to the Viper level. If you are interested in examples, you can find them inside the experiments/test-suites/viper_datatypes directory.

3.3.4 Matching Program States

One of our extension goals for this thesis was the support to generate program states that occurred earlier in the trace of the counterexample. We did not work extensively on this task but we had some ideas on how to tackle this problem. If someone wants to continue this work, here are some ideas.

The main idea is that given a counterexample we can go back to an older state in the execution trace if we take a subset of the constants (and heap entries in the SymbExLog) and then start the counterexample generation process for this subset of constants, functions and SymbexLog entries. Here is a simple example to illustrate the point:

```
1  method test()
2  {
3      var x: Int
4
5      x := x + 2
6      // user requests the state here
7      x := x + 2
8      x := x + 2
9      assert x != 7
10 }
```

The simplified SMT2 code for this Viper method looks like this:

```
1  (declare-const x@6 Int)
2  (declare-const x@7 Int)
3  (declare-const x@8 Int)
4  (declare-const x@9 Int)
5
6  (assert (= x@7 (+ x@6 2)))
7  (assert (= x@8 (+ x@7 2)))
8  (assert (= x@9 (+ x@8 2)))
9
10 (assert (not (not (= x@9 7))))
```

The requested state would than mean that the user is interested in the value from `x` at constant `x@7`. We would therefore remove the variables `x@8` and `x@9` from the Z3 model and the generate the counterexample.

There are multiple problems that need to be addressed. First we have to find this subset that corresponds to the right program state. Another could be branching, maybe the counterexample trace did not even reach the requested state or Silicon did some pre-calculations and removed it. This would mean that there are no constants that represent the requested state. Here is an example:

```
1  method test()
2  {
3      var x: Int := 1
4
5      x := x + 2
6      // user requests the state here
7      x := x + 2
8      x := x + 2
9      assert x != 7
10 }
```

The generated SMT2 code for this example looks like this:

```
1  (assert (not false))
```

No constants for `x` where created (as it is not needed) but we cannot generate the requested state with this technique if there are no constants.

This concludes the work we have done on the counterexample generation during this thesis. There are most likely more open problems than we encountered. As our approach was an experimental one and we did not argue over all the possible Viper and SMT2 configurations.

Chapter 4

Implementation

This chapter will describe the implementation of the counterexample generation process we created in this work. Not every idea discussed in the previous chapter but rather a baseline process has been implemented. Also we created a first version of the visualizer, that is a proof of concept for the debugging support.

4.1 Overview

Our software is written in Python, along with some bash scripts. It can be found in the `src` directory of our repository [13]. In this directory you will find the following files/directories:

- `counterexample_generation`: Python project that implements the "Counterexample Generator", the "Visualizer" and the second part of the Input Preparation Phase from the Approach chapter.
- `execute_viper_client.sh`: A Bash script that facilitates the interaction with the `viper_client`. The `viper_client` itself starts the verification task and it handles the communication with Silicon.
- `generate_counterexample.sh`: Main Bash script to start the counterexample generation and does the first part of the Input Preparation.

4.1.1 Getting started: Generate a Counterexample

To generate a counterexample for your failing verification, run the following script:

```
./generate_counterexample.sh "ViperFile" ["Output Directory"]
```

- `ViperFile`: Path to the Viper file

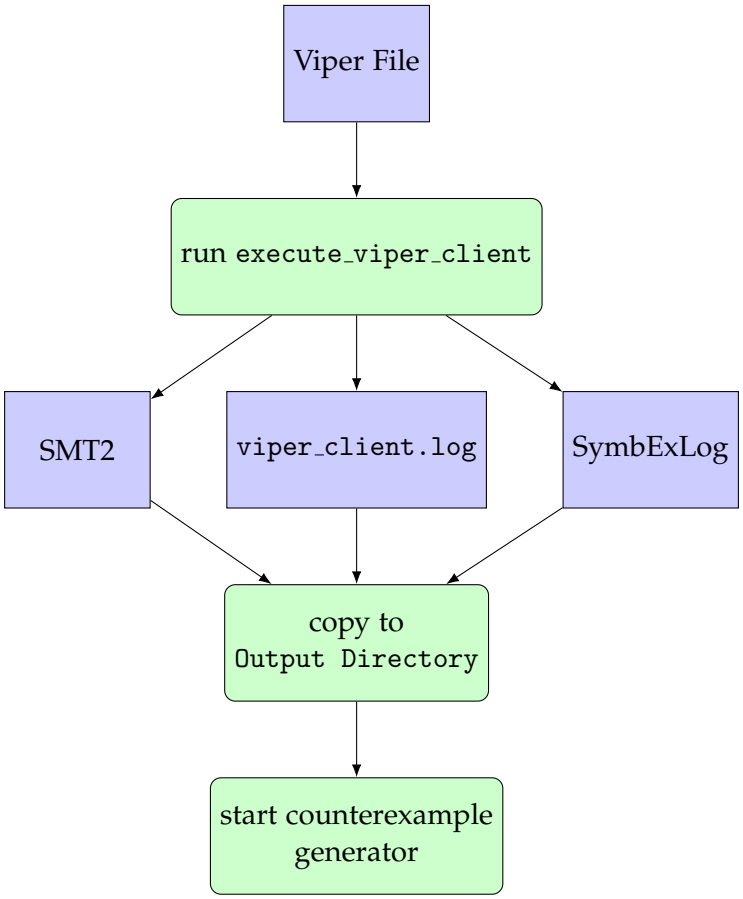

Figure 4.1: First part of the input preparation phase

- `Output Directory`: Path to the output directory, [optional, default: `./out`]

4.2 Input Preparation

In the first part of the input preparation phase the Viper file will be sent to Silicon (via the `viper_client`) and it creates the SMT2 File, SymbExLog and the `viper_client.log` containing all the failing methods with the error messages. These three files will then be moved to the `Output Directory` and the Python program will be started (see figure 4.1).

The second part is handled by the Python program. At that stage, the files are split into the method-specific sections. We want an SMT2 file for each failing method. We can then use Z3 to generate a model for the method specific SMT2 file to get the third input we need for the counterexample generation. The current implementation does not split the SymbExLog into

different files, here we load only the required parts if we need some information. We create a subdirectory for each failing method and copy the files to the subdirectory.

4.2.1 Post-processing the SMT2 file

As a final step before the counterexample generation can be started, we have to run a post-processing routine on the SMT2 file. The sources on this are contained in the `postprocessing` package. First all the `(get-model)` statements are removed and then add this statement after the final `(check-sat)`. We do this because we are only interested in the model in the failing state.

Subsequently we remove unnecessary statements from the SMT2 file. The SMT2 file contains `(push)` and `(pop)` statements: Every time a `(pop)` is sent to the SMT Solver, it returns its internal state back before the last `(push)` happened. It basically unlearns all the assertions that happened in between these two statements. Hence we can remove all assertions that are in between some `(push)` and `(pop)` statements with the same index. This does reduce the execution time of Z3 as these assertions are meaningless to the final generated model. For illustration purposes a short example is given below: on the left side you see the original SMT2 input and on the right side the final SMT2 input after the unnecessary assertions where removed.

```
1  (push) ; 1
2  (push) ; 2
3  (assert (= (f x) 2))
4  (check-sat)
5  ; unsat
6  (pop) ; 2
7  (assert (= (f y) 3))
8  (check-sat)
9  ; unknown
10 (pop) ; 1
```

```
1 (push) ; 1
2 (assert (= (f y) 3))
3 (check-sat)
4 ; unknown
5 (pop) ; 1
```

4.3 Counterexample Generator

Now that we have completed the file preparation we are ready to have a look at the implementation for a single failing method. As we have split up the failing methods we can now solve every method independently.

4.3.1 Parsing

Beforehand we need to parse the input files. In the package `inputparsing` you can find all the different parsers we used during this thesis. In every parser file are also the model classes. The different parsers are:

- smt_parser.py
- symbex_log_parser.py
- viper_client_log_parser.py
- z3_model_parser.py

For the SymbExLog we also have an infix_to_smt_converter.py as some code expressions inside the SymbExLog are in infix notation and we have to convert those to SMT2.

4.3.2 Building the counterexample

After parsing all the inputs the preparation for the generation process is completed. The counterexample generation has been implemented in the following files:

- counterexample_generator.py
- counterexample.py
- function_application_graph.py
- forall_instantiations.py
- matching_trigger_candidates.py
- z3_solver.py

The generated counterexample consists of 4 objects. The first three are a list of store variables, a list of heap entries (heap fields, quantified permissions and predicate instances) and a list of all the user-defined function models. The fourth object is the generated function graph. Additionally the input files are also stored within the counterexample. The definitions can be found in the counterexample.py file. If you are interested in the generation of the store, heap or function objects, please refer to counterexample_generator.py. For the function graph look at the file function_application_graph.py and its dependencies.

4.3.3 Detailed explanations

Triggering of Universal Quantifiers

In Section 3.2.2 we only described how we could use the function graph and not how we get to all the different function applications. Also in that section, we stated that the best approach would be to get these from the SMT solver. But as we could not achieve this, via the SMT2 interface, we had to implement our own solution. In the following section we will discuss how we handled the triggering of the universal quantifiers. We used this to generate more function applications for the function graph.

The first set of function applications we obtain directly from the SMT2 file. To get the same set of function applications as the SMT solver, we would have to trigger the same universal quantifiers as the solver. This means just reimplementing what the SMT solver does. This is undesirable and also out of scope for this project. Therefore we created a lightweight solution that is able to trigger all available universal quantifiers based on a given ground set. If desired we could repeat this process, if we added the newly found applications in each iteration to the ground set and repeat the whole process. The question posing itself at this point clearly was: How many repetitions are required? The current implementation only runs the whole process once and does not repeat it.

We will explain the solution based on a simplified example. You can find the code in these files:

- `function_application_graph.py`
- `forall_instantiations.py`
- `matching_trigger_candidates.py`

Let us assume that we have three functions `f: Int -> Int`, `g: (Int x Int) -> Int` and `h: Int -> Int`, additionally we have the following universal quantifier:

```
1 (forall ((x Int) (y Int)) (!
2     (= y (g (f x) y))
3     :pattern ((f x) (g (h x) y))
4 ))
```

Inside the SMT2 file we found the following function applications:

```
1 (f 1)
2 (f 2)
3 (f 4)
```

```
1 (g 1 1)
2 (g (h 1) 1)
3 (g (h 3) 1)
4 (g (h 3) (f 4))
```

The first step is to check for all `f` and `g` function applications if they match the trigger structure. For `f` this is `(f x)`, as we only have syntactically correct `f` applications inside the SMT2 file (otherwise we would have crashed during the first execution of this SMT2 file) we can assume that every function application matches this trigger. For `g` this is different, the trigger has the form `(g (h x) y)`. Therefore the first argument has to be an application of the function `h`. After this filtering step we have the following candidates for our matching:

```
1 (f 1)
2 (f 2)
3 (f 4)
```

```
1 (g (h 1) 1)
2 (g (h 3) 1)
3 (g (h 3) (f 4))
```

The next step is to find all combination of candidates (where we always choose one candidate out of each group) that satisfy equality over the quantified variables. In our example the trigger was `(f x) (g (h x) y)`, therefore the argument of the candidate for the `f` function has to be identical to the argument of the `h` function. As we only have one `y`, we don't have any constraints on it and any value is sufficient.

To solve this problem, we translated all the remaining function applications to a set of assignment:

```
1 {x = 1}
2 {x = 2}
3 {x = 4}
```

```
1 {x = 1, y = 1}
2 {x = 3, y = 1}
3 {x = 3, y = (f 4)}
```

Our goal is now to select one set of assignments out of each group without getting any conflicts. The fastest solution we developed translates this problem to a SAT query and we had to create a new Z3 instance to solve this problem. The translation is relatively straightforward. For the above example it would look like this (we replaced `(f 4)` with a new integer `5`):

```
1  (and
2      ; choose one out of the first group
3      (or
4          (and (= x 1))
5          (and (= x 2))
6          (and (= x 4))
7      )
8      ; choose one out of the second group
9      (or
10         (and (= x 1) (= y 1))
11         (and (= x 3) (= y 1))
12         (and (= x 3) (= x 5))
13     )
14 )
```

Z3 will return us the model `x = 1` and `y = 1` for this query. Therefore we know that we should trigger the universal quantifier with the variables `x = 1` and `y = 1`. To get all solutions, we will assert the negation of this model (`(not (and (= x 1) (= y 1)))`) and then ask for the next. This process can be repeated until an UNSAT response is reached.

Simplified Function Graph

We added a simplified version for the function graph, as we wanted to remove as much Z3 and Silicon internal information as possible. In this version the result nodes for functions have been removed, if this result node is only used as an argument for another function. As an example, the call `f(g(2))` with `f(x) = x` and `g(x) = x + 1` in the normal version would look like this:

```
2->g->3->f->3
```

But in the simplified version the result node for the `g` function has been removed and the graph would look like this:

```
2->g->f->3
```

Filtering of the quantified permission

While in Section 3.2.3 it was discussed how the partial functions had to be filtered we will now demonstrate the implementation of this process. For simplicity's sake the example from Section 3.2.3 will be reconsidered. The permission expression we looked at was:

```
0 <= inv@9(r) ==> inv@9(r) < len(a@2) ?  W : Z
```

Our goal was to take the left side of this implication and check for each case of the partial function if it is true in the current model. To check this, we have to translate the expression from an infix representation to into an SMT2 expression (look at `infix_to_smt_converter.py` if you are interested in this part). The converted SMT2 expression reads:

```
(<= 0 (inv@9 r))
```

Again, here is the partial function we talked about:

```
1 $FVF.lookup_val($FVF<Int>!val!2) = {
2     $Ref!val!1 -> 3
3     $Ref!val!2 -> 5
4 }
```

If we now take the first case of this function and replace the `r` variable in the SMT2 expression with `$Ref!val!1` we get the following:

```
(<= 0 (inv@9 $Ref!val!1))
```

Requesting a validation from Z3 will result in an error, because the value `$Ref!val!1` is an internal value and we cannot use these internal values in queries. These are only allowed to appear in models.

To overcome this problem we used a combination of constant model values and our function graph. As every internal value Z3 presents us in the model corresponds to some constant or function application, we only have to find the suitable expression that is in the same equivalence class with `$Ref!val!1`. If we assume that our function graph contains the exact same function applications as Z3 internally, than we would have an expression for every internal value Z3 can present us. This means that during the creation of the function graph we create an inverse lookup dictionary (from model value to expression) and also fill this dictionary with the SMT2 constant and their model values. At the end we have a dictionary for every internal value Z3 presents us and we can map them back to an SMT2 expression. In our example the dictionary would give us the expression `(loc a@2 0)` for the internal value `$Ref!val!1`. The SMT2 expression to check if the left side of the implication is true would look like this:

```
(<= 0 (inv@9 (loc a@2 0)))
```

If this expression evaluates to true, this case is part of the final partial function.

4.4 Visualizer

In this section we will go over the three main parts of a counterexample (store, heap and functions) and talk about how they are visualized. For the visualization we used the graph tool `graphviz` [7]. The concept of this prototype is based on ideas from a previous Master Thesis written by Alessio Aurecchia [2]. You can find the implementation in the file `visualisation.py`.

4.4.1 Store

The easiest part is the store (example in figure 4.2); it displays nodes for each user-defined variable. This node label contains the name of the variable, the type and its model value. If the type is a reference (`$Ref`), an edge will be drawn to the heap entry that represents that memory location and the model value will not be displayed. In the current implementation this only works for simple permissions, not quantified permissions.

4.4.2 Functions

Every function that is in the counterexample functions list will be displayed. To do this, we add the functions as a box and every case of this function model to this box (we remove the `else` case). See Figure 4.3 for an example.

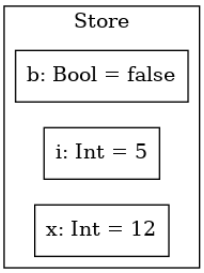

Figure 4.2: Example of a store visualization.

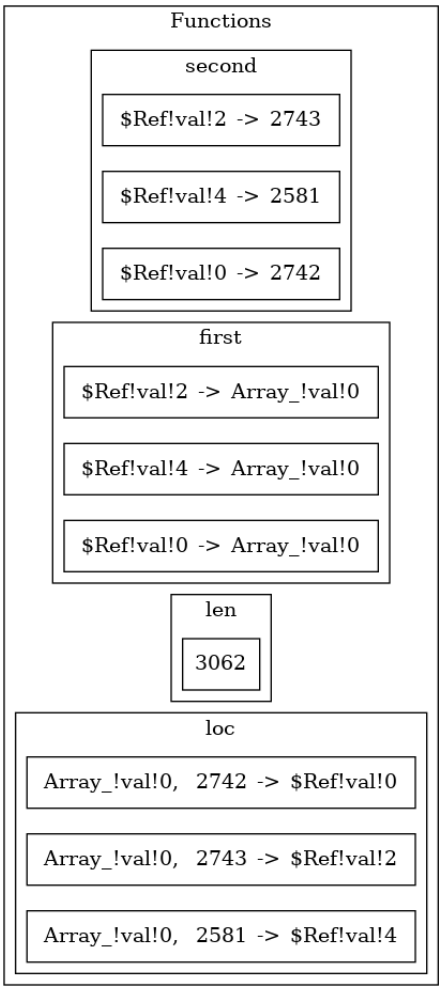

Figure 4.3: Example of the functions visualization.

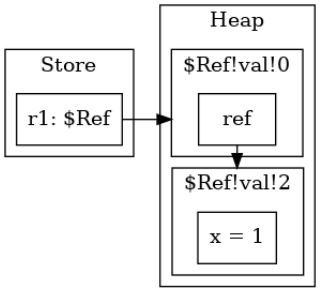

Figure 4.4: Example of a heap visualization.

4.4.3 Heap

For the heap we add every reference value (the model value of a reference variable; think of them as memory locations) as a box. The title of this box displays the reference variable and then a smaller box is added that represents the different fields of this memory location. Every field contains its name and model value (in the current implementation it does not contain a type). If the value of the field is a reference (`$Ref`) then it won't be displayed but instead an edge is added to the referenced heap entry (see 4.4 for an example).

Quantified Permissions

In Chapter 3 (Counterexample Generation) we talked about how we generate a partial function for quantified permissions. In the visualizer we will use this partial function and display it similarly to a normal function, but inside the heap box.

Predicates

If the counterexample contains predicate instances, the current implementation does add them as a single node to the heap box. It does not parse the arguments of this predicate, it will just display them to the user. It is a bare bones implementations, so a lot of improvements could be done on this subject.

Chapter 5

Evaluation

To evaluate what has been achieved by this project we will use our implementation of the counterexample generation and the prototype for the visualizer. We will look at different tests and check if the implementation can find a counterexample, if that counterexample is correct and finally if the visualizer can generate an image that visualizes all the important part of this counterexample. The last step will then be to compare our results of this theses with the project description that was created at the start of this project.

5.1 Example Set

The examples created during this thesis focused on simple test cases for the different Viper features. We used these examples to build the counterexample generation process. The resulting examples are relatively small and mostly contain only the feature that is being put to the test. Later we built some examples that combine multiple features but we don't have an example for every combination. Currently our example set contains 35 test cases, which should be enough to discuss the main use cases of the tool. The list of examples is given in Appendix A.1.

5.2 Case Study

In the following sections the counterexamples were generated with the default settings of the software. Therefore the simplified function graph is enabled. We will now go over different examples from the example set and analyse the generated counterexamples for each of these examples.

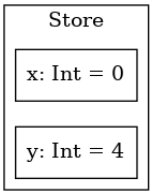

Figure 5.1: The visualisation of the generated counterexample for the first example.

5.2.1 Arithmetic Example

Let us start with an arithmetic example that only contains two store variables and has a simple structure.

```
1 method test(x:Int, y:Int)
2 {
3     assert 2*x*x + 3*x + 10 > y*y
4 }
```

This example has a method `test` with two integer parameters, `x` and `y`. The method consists of a single assertion with an arithmetic expression. This expression does not hold for every possible assignment for `x` and `y` and therefore we get a verification failure. The generated counterexample is also straightforward and only contains the two values for the parameters (see Figure 5.1). It contains the assignments `x = 0` and `y = 4`; if we use these assignments in the original formula the result will be `10 > 16` which is indeed not correct and therefore the counterexample is correct. The visualisation does also only contain the relevant information a user needs and is small enough to show inside an IDE.

5.2.2 Reference Field Example

Let's now have a look at heap permissions and heap fields that store references.

```
1 field x: Int
2 field ref: Ref
3
4 method test(r1: Ref)
5     requires acc(r1.ref) && acc(r1.ref.x)
6 {
7     assert r1.ref.x == 0
8 }
```

This example contains two fields, an integer one (`x`) and a `Ref` (`ref`), and a method `test`. The method has a `Ref` parameter `r1` and requires access to `r1.ref` and `r1.ref.x`. The body of this method then only has an assertion

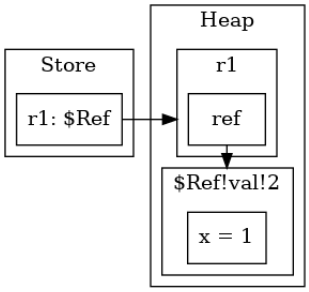

Figure 5.2: The visualisation of the generated counterexample for the reference field example.

that checks that the value of this last field `r1.ref.x` is equal to `0`. As this does not have to be true, the verification fails and the system will therefore generate a counterexample. You can find the visualisation of the generated counterexample in Figure 5.2. It generated a `r1.ref.x` field permission with the value of 1 and additionally the references `r1` which points to `$Ref!val!0` and `$Ref!val!0.ref` which points to `$Ref!val!2`. The last reference in this chain, `$Ref!val!2`, has an assignment `x = 1`. As the value was not `0`, this counterexample is correct.

5.2.3 Increment Function

So far the store and heap have been treated. In this next example we will discuss an example that contains a function. This will enable the generation of the function graph and the function section in the visualization.

```
1  function inc(x:Int): Int
2      ensures result == x + 1
3  {
4      x + 1
5  }
6
7  method test(x:Int)
8  {
9      assert inc(x) == 2
10 }
```

The example has a `test` method with a single integer parameter `x` and an `inc` function. As the assertion `inc(x) == 2` of the method only holds true for `x = 1`, a verification failure will occur. Figure 5.3 displays the generated counterexample for this case. We can see that it generated a constant function model for the `inc` function with a value of 238. This image also contains a function application graph and this introduces a new, for the user unknown, identifier `$Snap.unit`. As the function is independent of the heap the snapshot is not used and therefore the default one `$Snap.unit` has been

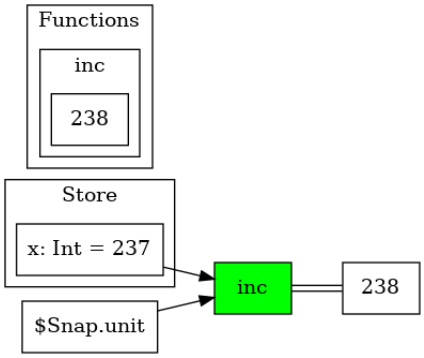

Figure 5.3: The visualisation of the generated counterexample for the increment example.

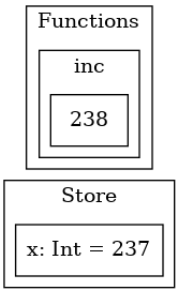

Figure 5.4: The visualisation without the function graph.

used. In case a user is not aware of snapshots, he will be irritated by its occurrence. One way to solve this problem is simply to remove the function graph from the final result, as depicted in figure 5.4. For this simple example this would be a reasonable solution, but if the examples get more complicated this could no longer work. Another way would be to remove the snapshot; as the `inc` function is not heap dependent this would work. But as soon as we have heap dependent functions, this solution would no longer be sound.

5.2.4 Array Maximum

As a last example for the evaluations we will look at an example that combines multiple features from Viper. It is a faulty implementation of a find-the-maximum algorithm in an array (loop through the array and find the biggest integer). It compares the current element with the previous element and not with the current maximum. Therefore the algorithm is not correct and we expect a verification failure.

```
1 // Removed the default Array definition
2
3 method find_max(a:Array) returns (max:Int)
```

```
4       requires access(a)
5       requires forall i:Int :: 0 <= i && i < len(a) ==> loc(
            a, i).val >= 0
6
7       ensures access(a) && untouched(a)
8       ensures len(a) == 0 ==> max == -1
9       ensures forall i:Int :: 0 <= i && i < len(a) ==> loc(a
            , i).val <= max
10  {
11      if(len(a) == 0)
12      {
13          max := -1
14      }
15      else
16      {
17          max := 0
18          var i:Int := 0
19          var last: Int := 0
20          while(i < len(a))
21              invariant i >= 0 && i <= len(a)
22              invariant access(a) && untouched(a)
23              invariant forall j:Int :: { loc(a, j) } j >= 0
                    && j < i ==> loc(a, j).val <= max
24              invariant i > 0 ==> last == loc(a, i-1).val
25              invariant i == 0 ==> last == 0 && max == 0
26          {
27              if(loc(a, i).val > last)
28              {
29                  max := loc(a, i).val
30              }
31              last := loc(a, i).val
32              i := i + 1
33          }
34      }
35  }
```

The main algorithm is located within the `find_max` method. As we pointed out earlier, this algorithm compares the current element with the previous one and not with the current maximum. Therefore if we call `find_max` on `[3,1,2]` it will output `max = 2` and will not find the correct `max = 3`. Therefore the verification fails as the loop invariant might not be preserved. If we now take a look at Figure 5.5, we can see that it also found this problem (albeit with other numbers). To facilitate the overview over the output, we removed the function graph from the visualisation for this report.

The analysis of this counterexample is a bit more complex than the previous ones as it contains a lot more information. If we start with the store variables, we can see that our array `a` has the value `Array_!val!0`, the `max` is 2748 and

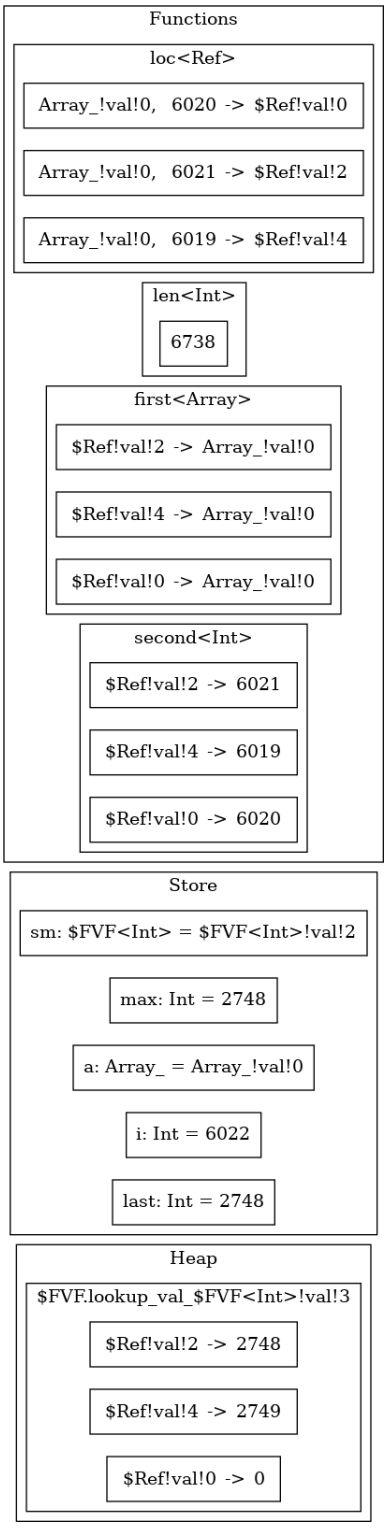

Figure 5.5: The visualisation of the generated counterexample for the Array Maximum example.

`i` is `6022`. If we continue with the model for the `loc` function, we can observe that the important indices are `6019`, `6020` and `6021` and the corresponding memory locations are `$Ref!val!4`, `$Ref!val!0` and `$Ref!val!2`. To get the values at these array locations we have to use the partial function generated for the quantified permission. In the current implementation it is named `$FVF.lookup_val_$FVF<Int>!val!3`. This function has exactly three cases, one for each memory location. Using this function we get to the following array values: `a[6019] = 2749`, `a[6020] = 0` and `a[6021] = 2748`. This is a correct counterexample to our initial Viper file.

In this example we can see even more new identifiers than in the previous one and we no longer have the option to just remove them, as in this example the new identifiers play an important role for the counterexample. It would be beneficial to have some sort of simplification steps on the counterexample, for more information about this idea see chapter 4.3.3.

5.3 Comparison to the Project Description

Now we will compare our results to the goals set in the project description. There we specified some core goals and a set of extension goals. As the main focus a of thesis lies on the core goals, we will start with those.

5.3.1 Core Goals

Build an example set

The first core goal was to build an example set, as described in this chapter we could achieve this goal and got an example set of 35 different examples that span different Viper features. It is not complete (and cannot be complete) but it contains a wide enough field of examples to talk about the general idea and if it would be possible to create such a counterexample generation engine.

Modelling in SMT2

During the time we wrote the project description we thought that we have to remove the quantified constraint. Therefore we included this goal to rewrite (or remodel) the SMT2 queries. One of the first results of this thesis was the realisation that this is not needed and we can directly reuse the queries generated by Silicon. This means that this step is required and the core goal can be removed (or simply it was achieved by the default SMT2 query).

Using Z3 as a model Finder

The next thing was using the Z3 SMT solver to generate the counterexamples. As the generated queries of a failing verification already gener-

ated a counterexample (but on the SMT2 code level) the first step was achieved early during the project. The harder part was creating a method that mapped this model back to the original source code. This could be achieved for some Viper features but not on all of them.

Alternative: using Z3 as an oracle

As we used Z3 as a model finder, we did not work on this goal.

Implementation

We created a prototype implementation for the counterexample generation and the visualization of the counterexample. This prototype was a standalone software and not integrated into the original debugger of Viper. So there is still work to do, to create a real implementation that is tightly integrated into the Viper tool-chain and also the VS Code IDE.

5.3.2 Extension Goals

Matching Program States

The first extension goal was to support finding program states from earlier in the failing trace. For example given the program state of the failing assertion, what was a possible state at the beginning of the method (or any other possible state in this method). We talked about this in section 3.3.4 but did not implement it in the prototype.

User-provided constraints

As we did not wrote an IDE extension that implemented our way of generating a counterexample we also did not support user-provided constraints. As described in the extension goal, an additional constraints is just an additional assumption on the Viper level and therefore Silicon should do the transformation to the SMT2 level. The counterexample generation engine at the bottom of the tool-chain would then not even recognise that some of the assumption where user-provided and not already inside the Viper code.

Support for Carbon

During the thesis we focused on the support for Silicon. Therefore we did not work on this extension goal.

5.4 Comparison to other Tools

In the field of program verification they are many other ways of supporting the developer in case of a verification failure. The current implementation

of the Viper Debugger, like other Symbolic Execution based verifiers for example VeriFast [8], visualise the symbolic verification states. Additionally the Viper debugger has a counterexample generation engine based on Alloy. Alloy has a few disadvantages in comparison to the model generation with Z3 as it only works over a bounded search space (for example we have to limit the range of integers it considers) and it does not have a native theory support. But every improvement in this section is a positive advancement as the most used tool is just adding new assertions to the failing verification file to check if some expression is true.

Chapter 6

Conclusion and Future Work

The goal of this thesis was to generate counterexamples using Z3 as a model finder. A previous project by Alessio Aurrechia accomplished this using Alloy as an external tool. But Alloy has some serious disadvantages as it only support searching for counterexamples over some bounded space and it does not have native tool support for theories used by Viper. Therefore our goal was to remove the Alloy dependency and use the SMT solver that is also being used to do the verification task.

In the end we created a standalone tool that is able to create counterexamples given the a Viper source file. First Silicon translates it into SMT2 statements and Z3 can then create a model (a counterexample) for this program. The main task of this thesis was then to map this model (using SymbExLog and the SMT2 file) back to the Viper source code.

The tool does not support every feature of Viper, but for the features it supports and with a little understanding on how Silicon translates a Viper file into SMT2, it is a useful tool for Viper developers. The automatically generated counterexample does help debugging failing verification tasks. Even if the current implementation of the visualisation of the counterexample is just a proof of concept, it still manages to display the important parts.

As a conclusion we can say, that an IDE for program verification in the future must have support for counterexample generation and also some way of visualising it. This thesis has been a step into the right direction to achieve this goal for Viper and its integration into the VS Code IDE.

6.1 Future Work

As previously discussed, the current implementation is just a proof of concept, therefore there are a lot of open problems to solve:

- Integrate the software into the Viper Plugin for VS Code and reuse the data structure Silicon uses internally. At the moment our python software does the parsing of the input files and stores the data in its own model classes. It does not reuse already defined classes from Silicon/Viper (as these are written in scala). This is one of the important aspects to improve as otherwise the overhead to support and keep it up to date is enormous.
- Support more (at best all) of the different Viper features. Currently there are some features of Viper that are not directly supported. As most Viper features get translated to some SMT2 functions by Silicon, we support them because we know how to handle these generated functions but the final version should translate the models from the generated functions back to the Viper level and not just display these internal functions.
- Create a good visualizer. The current implementation of the visualizer is a simple one. It does work for small counterexamples but if the number of nodes get too big, the generated image gets messy. The lines overlap and the user cannot easily extract the information that the image should display. In a future version of the IDE, the generated counterexample should be interactive. There should be a way to hide some parts of the counterexample, move them around, rename them and even link them to the source code.
- Improve the communication between the SMT solver and the counterexample generation engine. As the SMT solver does most of the work, we should reuse a lot of things it already did (but normally does not expose). For example, currently we manually do a quantifier instantiation to get all the function applications on the python side. A closer integration with Z3 should enable us to ask Z3 directly, what function application it knows about. This would save us some calculation time and even more important, it would be a cleaner solution in the end.

Appendix A

Appendix

A.1 Examples

Nr	Test Group	Main Feature	Test case	Description	Verification Failure	CE found	CE correct	Time [s]	Visualization
1	array_access	QP	access	array with stored indices, try to access a[a[0]]	yes	yes	yes	1.6575	ok
2		QP	access_len0	array with stored indices, len(a) == 0	no	no		0.8788	
3		QP	access_len0_loc5	array with stored indices, len(a) == 0, acc a[a[5]]	yes	yes	yes	1.6272	ok
4	snapshots	Snapshots	snapshot_structure	simple testcase to check how snapshots are built	yes	yes	yes	3.1126	ok
5	max	QP	0_max	wrong max algorithm	yes	yes	yes	1.1034	too big, function graph not helpful
6		QP	1_max_len3	with len(a) == 3	yes	yes	yes	4.2902	too big, function graph not helpful
7		QP	2_max_len2	with len(a) == 2	yes	yes	yes	5.6563	too big, function graph not helpful
8		QP	3_max_len2_fix	fixed an error in loop invariant	no	no		3.1847	
9		QP	4_max_fix	back to the original (with the found fix)	yes	yes	yes	4.6683	too big, function graph not helpful
10	nested_function_calls	Functions	nested_add	nested function applications	yes	yes	yes	1.306	ok
11	permissions	Permissions	full_permission	full permission ref field	yes	yes	yes	1.2171	ok
12		Permissions	fractional_permission	fractional permission ref field	yes	yes	yes	1.1034	ok
13		Permissions	minimal	minimal testcase with a ref field	yes	yes	yes	1.2245	ok
14		Permissions	ref_fields	field with type reference	yes	yes	yes	1.1388	ok
15		Permissions	ref_returned_by_function	function that returns a reference	yes	yes	yes	1.3322	ok
16	predicates	Permissions	predicates	small testcase with a predicate	yes	yes	yes	1.2112	ok
17	basics	Store Variables	ifelse	an if/else block	yes	yes	yes	1.1369	ok
18		Store Variables	if	a single if block	yes	yes	yes	1.2368	ok
19		Store Variables	arith	arithmetic expression	yes	yes	yes	1.1255	ok
20		Store Variables	multiple_errors	multiple failing methods	yes	yes	yes	3.4442	too big
21	viper_datatypes	QP	sets_1	Set example 1	yes	yes	yes	1.4329	function graph not helpful
22		QP	sets_2	Set example 2	yes	yes	yes	1.3844	function graph not helpful
23		QP	sequences	sequence example	yes	yes	yes	2.0943	function graph not helpful
24		QP	multisets	multiset example	yes	yes	yes	1.4363	function graph not helpful
25	two_arrays	QP	two_arrays	two int arrays	yes	yes	yes	17.0595	too big, function graph not helpful
26		QP	different_array_types	two arrays with different datatypes	yes	yes		45.3618	too big, function graph not helpful
27	different_max	QP	max	another implementation of the same max algo from max	yes	yes	yes	5.4568	ok
28	quantified_permissions	QP	quant_permission	first example with a quantified permission	yes	yes	yes	1.8506	ok
29		QP	arraySize	assertion on a len function	yes	yes	yes	1.3687	ok
30		QP	simple_array	assertion on the content of the array	yes	yes	yes	1.6374	ok
31	introduction_max	QP	intro_max	max algorithm from the introduction chapter	yes	yes	yes	3.1521	ok
32	functions	Functions	state_independent	function without a state dependency	yes	yes	yes	1.1503	ok, function graph doesn't really help here

33		Functions	state_dependent	function with a state dependency	yes	yes	yes	1.8818	ok, snapshot should need an alternative visualization
34		Functions	recursive_function	a recursive function	yes	yes	yes	1.8413	too many function cases
35	linked_list	QP	linked_list	a linked list example	yes	yes	yes	4.6541	ok

Bibliography

[1] Alloy. alloytools.org. http://alloytools.org/. [Online; accessed 04-Mai-2019].

[2] Alessio Aurecchia. Visual debugging for symbolic execution. Master's thesis, ETH Zürich, 2018.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The smt-lib standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[4] Ivo Colombo. Debugging symbolic execution. Master's thesis, ETH Zürich, 2012.

[5] Leonardo de Moura and Nikolaj Bjørner. Z3 smt solver. https://github.com/Z3Prover/z3/. [Online; accessed 14-Mai-2019].

[6] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[7] Graphviz. graphviz.org. https://www.graphviz.org/. [Online; accessed 08-Mai-2019].

[8] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In Kazunori Ueda, editor, *Programming Languages and Systems*, pages 304–311, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[9] Bart Jacobs, Jan Smans, and Frank Piessens. Verification of imperative programs: The verifast approach. a draft course text, 2010.

[10] Ruben Kälin. Advanced features for an integrated verification environment. Master's thesis, ETH Zürich, 2016.

[11] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

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

[13] Cédric Stoll. Repository of our software. https://bitbucket.org/cstoll_ethz/viper_counterexample_generation/src/master/. [Online; accessed 18-Mai-2019].

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

SMT Models for Verification Debugging

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Stoll	Cédric

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Frauenfeld, 18.5.19	C. Stoll

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