Counterexample Generation in Gobra
Bachelor's Thesis - Description

Fabio Aliberti

Supervised by : Linard Arquint, Felix Wolf

March 2021

1 Introduction

Go [3] is a statically typed programing language, that offers features to write highly concurrent code. Thus it is the basis of many distributed systems such as Netflix [4] or SCION [9], for which it is vital to prove security properties and the absence of bugs. Conventional program testing leaves a lot of room for errors, since each testcase can only cover a single execution. In particular for concurrent programs, the number of thread interleavings increases exponentially. Instead of writing infinitely many testcases, we can use program verification to search for a formal proof. Gobra [1] is a deductive verification tool to verify Go programs. It takes a program with specification annotations as input and tries to prove its correctness. If verification fails, Gobra outputs the program locations indicating which assertions it could not prove. Assertions can fail for multiple reasons: (1) the code has unintended behavior, (2) the specification is too strict, and (3) the automation of the verification backend is insufficient. Identifying which case applies can be hard, in particular for more complex programs, where it can be non-trivial to figure out the underlying cause. We aim to address this issue, by generating counterexamples, displaying a value assignment for which an assertion does not hold. Especially when dealing with corner cases, it is helpful to get concrete values that are troublesome. Counterexamples also improve the verification workflow. The programmer can use counterexamples to evaluate assertions on a concrete value assignment and thus iteratively improve the specifications and code, until verification succeeds. The goal of this project is to add support for counterexamples to Gobra.

1.1 Context

Viper [8] is a verification infrastructure, that supports permission based reasoning. Gobra uses Viper, by translating the Gobra program into a Viper representation. Gobra then hands this Viper program to one of two backends: Silicon or Carbon. We will be focusing on Silicon, as previous work on counterexamples has been done for Silicon, namely in the context of Nagini [7] a verifier for Python, and Prusti, a verifier for Rust [2, 6]. The Silicon backend uses the Z3 [5] SMT solver to discharge Vipers proof obligations and produce counterexamples. Silicon then translates counterexamples produced by Z3 into a Viper representation that is not Silicon specific. So far, the translation includes the basic Viper types Bools, Ints, Refs, and Sequences. However, a translation for user-defined Viper types is still missing, as they have not been of interest for Viper programs encoded by Prusti. Since Gobra uses many user-defined types throughout the encoding, we aim to translate counterexamples for user-defined types to a useful Viper representation. For a useful representation, we might have to pass additional information about a user-defined type to the translation. Currently, Silicon instantiates user-defined types as abstract values. Consider the following user-defined type for Chars:

```
domain Char{
    function ofInt(x:Int):Char
    function toInt(c:Char):Int
    axiom axBound{
        forall c:Char::{toInt(c)}
            toInt(c) < 256 && toInt(c) >= 0
```

```
    }
    axiom axRel{
        forall c:Char :: { ofInt(toInt(c)) }
            ofInt(toInt(c)) == c
    }
    axiom axOrd{
        forall c:Char ,d:Char:: { toInt(c),toInt(d) }
            toInt(c) == toInt(d) <==> c == d
    }
}
method testGreater(c:Char,d:Char){
    assert toInt(c) >= toInt(d)
}
```

In Viper, user-defined types are represented by domains. A domain consists of the name of the type, here `"Char"`, as well as a collection of abstract functions and a set of axioms to give meaning to the functions. Even though, it is easy for a human to give meaning to such a domain, Viper only sees uninterpreted functions and axioms. To illustrate this we have a look at the method `testGreater`, that contains a single assertion statement. Currently, Silicon generates the following counterexample for `testGreater` when executing the `--counterexample mapped` flag, which uses the existing Silicon to Viper counterexample translation.[1]

```
assertion toInt(c)>=toInt(d) might not hold :
possible counterexample :
c <- Char!val!1
d <- Char!val!0
```

The output lists the violated assertion, and gives us an assignment of `c` and `d`. With this information we can only infer that `d` and `c` are two different Char instances. This is because Silicon's counterexample translation into Viper filters information deemed unimportant, and ignores the instantiation of the `toInt` function. The counterexample at the Silicon level does contain this information. However, it also contains a large amount of uninteresting information, which is used by the Silicon internal state. If we want to take a closer look at this internal state, we can do this by invoking Silicon with the `--counterexample native` flag.[2] At the Silicon level, the `toInt` function is instantiated as: `toInt(c) = if(c== Char!val!0) then 42 else 0`. Using this additional bit of information, we can output our counterexample as follows:

```
assertion toInt(c)>=toInt(d) might not hold :
possible counterexample :
c <- Char!val!1 where toInt(c) == 0
d <- Char!val!0 where toInt(d) == 42
```

This new output makes it clear why the assertion failed, specifically because the int value of `d` is larger than the int value of `c`. For our example, the output could even be further improved by using the actual Char that is represented by 42 (i.e "*"). Of course this is not always as simple or even desirable. As the translation happens on the Silicon level, we aim to not make it too Gobra specific. As mentioned above, Nagini can also generate counterexamples. Nagini directly translates Silicon counterexamples into Nagini counterexamples, without using Silicon's functionality to translate them into Viper counterexamples. Therefore we cannot reuse its code. Instead, we will use it as an inspirational source. On top of extending the translation from Silicon to Viper counterexamples, we will develop a translation from Viper to Gobra counterexamples from the ground up.

[1] Output may vary
[2] This output is omitted for the sake of brevity

2 Approach

2.1 Core goals

1 The first goal is to add a general framework to Gobra, for translating and outputting counterexamples to Gobra. This framework has to be robust enough that we can extend it step by step over the duration of this thesis. To sanely allow any follow up work, we preserve the variables at an AST level, rather than as simple strings. Next, we add support for Integers and exclusive Arrays, to test that our framework functions as intended.

2 Secondly, we will extend the translation of counterexamples from Silicon to Viper with the support for user-defined types. The challenge is to choose a suitable representation of domain counterexamples as well as developing a translation.

3 As our third goal, we will add translations and meaningful representations for built-in Gobra types, namely structs, interfaces, and pointers. Additionally, we will come up with an implementation concept for channels, quantified permissions, and first-class predicates.

4 Our fourth goal is to evaluate our support for counterexamples. We will perform a qualitative assessment by comparing the verification of Gobra with and without the counterexample generation enabled. Furthermore, we will get an assessment of the amount of redundant information by manually inspecting the counterexamples produced for specific inputs. We have observed that in practice, Gobra programs do not always generate Z3 counterexamples. Fixing this issue completely is beyond the scope of our core goals, but we will investigate for which programs Z3 fails to produce counterexamples.

2.2 Extension goals

1 The first extension goal is to add support for more gobra features, namely Channels, quantified permissions, and first-class predicates. These features are less important than the features listed in core goal three but they are still extensively used in Gobra.

2 Secondly, we reduce the size of counterexamples on the Viper level or in Gobra. We do this by filtering unnecessary information like constants or variables that do not impact the failed assertion. However, it is not trivial to identify which state is unnecessary. One approach might use a static analysis to solve the issue.

3 Lastly, we add counterexample support to the existing Gobra IDE [10] by embedding counterexamples into error messages. We will explore alternative ways to visualize counterexamples, such as providing the value of a variable when hovering over it.

References

[1] Gobra. `https://github.com/viperproject/gobra`.

[2] Prusti. `https://github.com/viperproject/prusti`.

[3] The go programming language. `https://golang.org/`, (visited: February 2021).

[4] Netflix. `https://www.netflix.com`.

[5] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 337–340. Springer-Verlag, 2008.

[6] Cedric Hegglin. Counterexamples in prusti. Bachelor's thesis, ETH Zürich, 2021.

[7] M.Eilers and P.Müller. Nagini: A static verifier for python. Technical report, ETH Zürich, 2018.

[8] 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.

[9] A. Perring, P.Szalachwski, R.M. Reschuk, and L. Chuat. Scion: A secure internet architecture. publication, ETH Zürich, 2017.

[10] Silas Walker. Ide suport for a golang verifier. Bachelor's thesis, ETH Zürich, 2020.