Recovering from Verification Failures

Bachelor Thesis Project Description

Josua Stuck
supervised by Dr. Malte Schwerhoff

February 2021

1 Introduction

Viper [4] is a collection of software verification tools, as well as a permission based intermediate verification language, developed at ETH Zürich. There are several Viper frontends for high-level languages such as Python or Rust, as well as two different verification backends. The Carbon backend uses verification condition generation, while Silicon [6] is based on symbolic execution. Both backends ultimately use the Z3 SMT solver to verify resulting proof obligations. In addition to using the provided frontends, users can also directly encode verification problems in the Viper language. This thesis is concerned with adapting the Silicon backend.

2 Background

2.1 Separation Logic

Separation logic [5] (an extension of Hoare Logic) is a program logic for reasoning about programs that manipulate pointer data structures . Its computational state comprises a *store* (a mapping from variables to values, including memory addresses) and a *heap* (a mapping from addresses to values). A separation logic specification $[P]C[Q]$ (where P is a precondition, C a program, and Q a postcondition) is to be read the following way: if C is given a heaplet (a portion of the heap) that satisfies P, it will never access heap outside of P, and return a heaplet that satisfies Q. Separation logic introduces the *separating conjunction*, written $*$ (&& in Viper). A heaplet h satisfies $P * Q$ if it can be split into two disjoint heaplets h_p, h_q that satisfy P and Q respectively. Note that $P \Rightarrow P * P$ is not sound. Based on this conjunction, separation logic supports an inference rule called the *frame rule*, which allows to frame R (the portion of the heap that is not modified by C) across C.

$$\frac{[P]C[Q]}{[R * P]C[R * Q]} \text{ C doesn't assign to R's free variables}$$

The frame rule allows to reason about portions of the heap, that is, if a program safely executes with some heap, it will also safely execute with an extension of that heap.

2.2 Fractional Permissions

Permissions are a way to enable framing. Accessing a heap location requires permissions to it. Holding permissions to a location allows the knowledge about that location's value to be framed across e.g. a method call. Fractional permissions [2], by allowing to split up permissions, are more powerful than all or nothing permissions, as they allow to differentiate between read and write permissions. A state with fractional permission (any fraction $0 < p < 1$) to a location allows to read from this locations, whereas writes require full permission (a permission of 1).

2.3 Pure and Impure Assertions

Impure assertions are assertions that add or remove permissions from the state. All other assertions are pure. Because an impure assertion adds or removes permissions, the identity $P \Leftrightarrow P * P$ does not hold for such an assertion P, whereas this would hold for a pure assertion.

2.4 Symbolic Execution

The general idea of symbolic execution [3] is to execute a program using a *symbolic state* that contains symbolic rather than concrete values. Each executed statement updates the symbolic state according to its (abstract) semantics. A method implementation can be verified by setting up a symbolic state such that the method's precondition holds, symbolically executing the method, and then checking whether the postcondition holds (in the symbolic state resulting from the symbolic execution). Silicon uses Smallfoot-style symbolic execution, named after Smallfoot[1] – the first automated verifier for separation logic. Such verifiers separate their symbolic state into *path conditions* and a *symbolic heap*. The path conditions correspond to assertions expressed in first-order logic, for example, equalities between symbolic values, whereas the symbolic heap corresponds to assertions that cannot be expressed directly in first-order logic, such as permissions to fields or predicate instances. Smallfoot-style verifiers typically branch over conditionals such as `if-then-else` statements. That is, at such a statement, the `then` branch (as well as the rest of the program) is executed under the assumption that the condition is true, and the `else` branch (as well as the rest of the program) under the assumption that it is false. Because of the two resulting branches, everything after the conditional is symbolically executed twice.

2.5 Viper

Viper is based on separation logic with fractional permissions. Below, an example Viper program is given. The *accessibility predicate* `acc(x.f, p)` denotes permissions `p` to a location `x.f`, where `x` is a reference and `f` is a field. Permissions to resources can be obtained from impure assertions, such as the precondition `acc(x.f, p)`. Similarly, permissions can be given up via impure assertions, such as `acc(x.f, p)` in the `exhale` statement on line `11` in the example below.
The assertion `x.f > 3` on line `6` is pure, as it only *accesses* (and therefore requires permissions to) resources from the heap, but does not *alter* permissions to that location.

```
1: field f : Int
2:
3: method example(x: Ref){
4:     inhale acc(x.f, 1/3)
5:     if (x.f > 3){
6:         assert x.f > 3
7:     } else {
8:         assert acc(x.f, 2/3)
9:     }
10:    assert false
11:    exhale acc(x.f, 1/3)
12: }
```

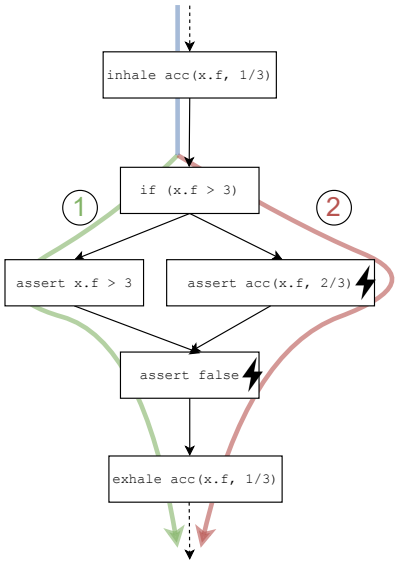

Figure 1: Silicon branches at the `if` statement, and verifies path 1 before path 2, reports an error at line 10, stops, never verifies path 2, and thus never reports the error on line 8.

3 Project

Currently, the Silicon verifier stops the verification process of the current verification unit (e.g. a method) as soon as it encounters an error (e.g. the `assert false` statement on line 10 of Figure 1). The goal of this project is to enable Silicon to recover from verification failures, such that more problems can be found in one verification run. From a user's standpoint, the current behaviour is not ideal for multiple reasons. For one, it requires the user to fix verification failures one by one, only being able to see the next error once the previous error is fixed. Furthermore, there are cases where the verifier seemingly doesn't catch an error.

In the example above, the `assert acc(x.f, 2/3)` statement on line `8` does not hold (only 1/3 permissions are held), but the current version of Silicon only reports that the assertion on line `10` might not hold. The reason for this is that Silicon branches at the `if` statement on line `5`, and first verifies the `then` branch (under the assumption $x.f > 3$). On line `10`, Silicon stops because the `assert false` statement does not hold. Therefore the `else` branch, which has a failing assertion on line `8`, is never symbolically executed (and the failing assertion is seemingly skipped).
This program also raises the question how such cases should be handled by the verifier. A naive approach to error recovery would lead to the error on line 10 being reported twice (or more if there are several conditional branches), which can clutter the output. On the other hand, if such an error occurred only on some branches, it would certainly be interesting to know under which branch conditions the error occurred.

Enabling Silicon to recover from verification failures will be achieved by assuming that the failing assertion held (and therefore adapting the verification state correspondingly), before continuing verification. Depending on the type of the failed assertion this is more involved: for a pure assertion P (for example `assert x.f > 3` on line `6`), if it failed, it would be sufficient to inhale P (here $x.f > 3$) before continuing the verification. For an impure assertion (for example `assert acc(x.f, 2/3)` on line 8) however, it would be unsound to inhale `acc(x.f,2/3)`. This would mean adding 2/3 permissions to the already held 1/3 permissions from line `4`, resulting in full permissions (write access). A sound approach would be to only add the missing permissions (1/3) such that the finally held permissions sum up to 2/3.

4 Goals

The goal of this project is to enable Silicon to recover from verification failures, to allow the verifier to find further problems. Below, this goal is broken down into smaller sub-goals.

4.1 Core Goals

- Create a collection of example programs that would benefit from error recovery. This can be done by getting examples from the user base (i.e. GitHub issue tracker), as well as creating new examples. The examples can then also be used for the performance evaluation.
- Improve Silicon's architecture such that one error per path can be reported.
- Improve error recovery by enabling the verifier to continue after a *pure* assertion failed.
- Design a way to handle assertions that fail on multiple branches. That is, determine how they should be reported (just once, or multiple times?; under which conditions?).
- Implement the designed handling of assertions that fail on multiple branches.
- Evaluate this version of Silicon with the evaluation scheme below.
- Design a solution to recover from failed *impure* assertions.

4.2 Extension Goals

- Implement the designed solution for impure assertions.
- Evaluate this version of Silicon with the evaluation scheme below.
- Investigate how joining after branches could be supported, estimate the necessary efforts, and attempt an experimental implementation, if time permits.

4.3 Evaluation Scheme

Both versions are used to verify a method with N failing assertions. Because old Silicon stops after each error, it will be run N times, but each time the error discovered in the last run will be fixed. New Silicon (the version that is to be evaluated) will be run just once. The verification times for old Silicon will be summed up and compared to new Silicon's verification time. The goal here is to not have a longer runtime with the new version of Silicon (compared to the summed up runtime of old Silicon).

References

[1] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 115–137, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[2] John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis*, pages 55–72, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[3] Sidney L. Hantler and James C. King. An introduction to proving the correctness of programs. *ACM Comput. Surv.*, 8(3):331–353, September 1976.

[4] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[5] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, page 55–74, USA, 2002. IEEE Computer Society.

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