Early Bug Detection in Viper Programs

Master Thesis Project Description

Thibault Dardinier
Supervisors: Gaurav Parthasarathy, Prof. Peter Müller

October 2019

1 Introduction

Various formal verification techniques can be used to automatically verify the absence of errors in programs. This provides an advantage over testing approaches, namely the guarantee that a program is correct for any possible execution. However, such approaches often require a user to provide additional specifications (such as loop invariants) to guide the verification, which places a burden on the user. When no specifications are provided, verifiers usually report potential errors which are not actual errors, hence lowering confidence in error reporting.

Users might want to learn quickly (without providing too many specifications) and with high confidence whether a program is incorrect. This would speed up the development and verification process by only having to provide specifications when one is fairly certain that the program is correct. Stratified inlining [1] is an approach which achieves this by iteratively performing static inlining of loops and recursive calls up to a bounded depth, in order to detect errors. The main goal of this thesis is to develop a technique, such as stratified inlining, to detect bugs early in Viper [2], which is a verification infrastructure for permission-based reasoning. One of the interesting challenges to overcome is that the effect of standard static inlining is unclear in Viper. Consider the following example, where the *while* loop is unrolled twice:

Listing 1: Initial code

```
1  var x: Ref
2  inhale acc(x.f)
3  x.f := 0
4  var i: Int := 0
5  while (i < 1) {
6    assert x.f == 0 // Is this a bug?
7    i := i + 1
8  }
```

Listing 2: After loop unrolling

```
1  var x: Ref
2  inhale acc(x.f)
3  x.f := 0
4  var i: Int := 0
5  if (i < 1) {
6    assert x.f == 0
7    i := i + 1
8    if (i < 1) {
9      assert x.f == 0
10     i := i + 1
11     while (i < 1) {
12       assert x.f == 0
13       i := i + 1 }}}
```

At first glance, one would expect these two programs to be functionally equivalent. Viper reports a potential error (due to insufficient permission to $x.f$) on line 6 of the initial code (listing 1), but

is able to verify the code after loop unrolling since the rest of the loop on line 11 (listing 2) will never be reached. The reason is that the Viper verifier does not take into account permissions held outside the loop, unless explicitly specified by the loop invariant. Therefore, to be able to verify the initial code, the programmer needs to add the invariant *acc(x.f)*. Should line 6 of the initial code be considered as an actual bug due to insufficient permission? To answer this question, one should define the role of a loop invariant.

The standard view would be to consider the loop invariant as an overapproximation of all states which reach the evaluation of the loop condition. This would mean that the loop invariant tracks only a subset of the overall permissions, in this case validating the interpretation that the assertion on line 6 of the initial code is correct. However, this view is inconsistent with respect to certain features. The reflective *perm* function is such a feature: *perm(x.f)* evaluates to the amount of permission the current method holds to the specified location (*x.f* in this example). Consider the following example, in which both code snippets are successfully verified by Viper:

Listing 3: Full permission

```
1  var x: Ref
2  inhale acc(x.f)
3  var i: Int := 0
4  while (i < 10)
5  invariant acc(x.f) {
6    assert perm(x.f) == 1/1
7    i := i + 1
8  }
```

Listing 4: No permission

```
1  var x: Ref
2  inhale acc(x.f)
3  var i: Int := 0
4  while (i < 10)
5  invariant true {
6    assert perm(x.f) == 0/1
7    i := i + 1
8  }
```

The only differences between these two code snippets are the invariants (line 5) and the assertions (line 6). Here, perm(x.f) evaluates to *precisely* the permission specified by the loop invariant. Therefore the two loop invariants cannot be interpreted as overapproximations of reachable states, but actually lead to clearly incompatible programs. In general, tracking a subset of the permissions enables the verification of programs which would not necessarily verify when holding a larger subset of permissions. One challenge here is to determine the implications for a technique like stratified inlining, which does not consider specifications.

Similar fundamental challenges show up when inlining method calls. The objective of this thesis is to analyze the challenges and to potentially clarify parts of the Viper semantics, in order to design and implement a technique enabling efficient early bug detection in Viper.

2 Core Goals

2.1 Exploration

The first task of this thesis is to extensively explore variations of the Viper semantics, while understanding their dependencies on user-provided specifications and their relationships to the Viper semantics as interpreted by the Viper verifier. It will be necessary to specify and define what should be the "correct" behavior of Viper regarding bugs, thereby clarifying some parts of the semantics. It will also be necessary to explore other Viper features that could be challenging for stratified inlining.

2.2 Design

Based on the previous exploration, a technique should be designed to enable the early detection of bugs in Viper programs. The soundness and completeness of this technique should also be clearly defined.

2.3 Implementation

This technique should then be implemented for the Viper infrastructure. There are several possibilities to achieve this:

1. Translating a Viper program to a set of Viper programs.
2. Modifying the Carbon back-end (for example, by reusing Boogie's built-in stratified inlining algorithm).
3. Modifying the Silicon back-end.

2.4 Evaluation

It is necessary to evaluate the implemented technique in terms of performance and usefulness, to ensure its relevance as a Viper feature.

2.5 Front-End Interface

An important objective of Viper is to bring formal verification to programmers. The last core goal is to evaluate the challenges of leveraging the implemented technique to bring the early bug detection feature to a front-end interface, and to design a technique to achieve this.

3 Extensions

Several extensions to this thesis are possible:

1. Optimizing the performance of the implemented technique.
2. Dealing with some sources of incompleteness such as loops with input-independent number of iterations. Using the stratified inlining technique with a bound smaller than this number of iterations will always result in incompleteness of the technique.
3. Implementing the early bug detection feature for a front-end interface, as designed in subsection 2.5.
4. Generalizing to general (reducible) control flow graphs (for instance, enabling the use of *goto* instructions).

References

[1] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 427–443, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[2] 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.