Adding Debugging Functionality to Viper

Practical Work Description

Andrea Keusch

Supervised by Prof. Dr. Peter Müller, Dr. Marco Eilers, Lea Salome Brugger

September 22, 2023

1 Introduction

Viper [4] is an intermediate language for program verification, the process of formally proving the correctness of programs with respect to a given specification. Viper has two backends: a Symbolic Execution backend (Silicon) and a Verification Condition Generation backend (Carbon). We will work on the Silicon backend.

Silicon. The input to Silicon [5] is a Viper program whose parts are then verified individually using Symbolic Execution [2]. This verification technique steps through the program as during program execution but keeps a so-called symbolic state (instead of a concrete state). Each variable is assigned a concrete value, if it is known, or a symbolic expression otherwise. Additionally, it stores a path constraint, which is the collection of all preconditions, assumptions, branch conditions and other constraints that are known to hold. Every concrete execution of the program satisfies this path constraint. Therefore, if one can prove that a statement is implied by the path condition, then it is proven that the statement holds for every execution. Whenever an assertion (or invariant or postcondition) is encountered, Silicon tries to prove it under the assumption of the path condition. In order to do so it invokes an SMT solver (Z3 [3] by default). The output of Silicon is either "verified", meaning that all assertions have been proven successfully, an error for each failed assertion or a timeout.

Debugging Viper. If not all assertions can be proven, the user has to find the bug and resolve it somehow. To do so efficiently, information about the error and tools that enable analysis are needed. So far, there exist only a handful of debugging techniques. First, the user gets some hints about the error, for example, it states at which point an invariant might not hold (before the loop, after each iteration or after the loop). Additionally, Silicon can provide a concrete counterexample, which can be helpful. But there is nothing else, such that users have to resort to some tricks in order to analyze parts of the code, for example by adding additional assumptions and assertions, or commenting out parts of the code.

However, there are several issues with these techniques. First of all, the user does not get enough information to analyze the error properly. The only way to get more information is to apply the aforementioned techniques through trial and error. After each modification of the code, the verification has to be rerun, which can take several minutes for very large programs. This debugging process can be frustrating and slow. Another big issue is that the modifications have to be undone before doing the full proof. Otherwise, the verification might succeed, but the proven statement is not the desired one. For all these reasons, it is essential to provide proper debugging tools that do not modify the Viper code and enable the user to get all information about the error with little effort.

2 Project Goals

2.1 Main Goals

The goal of this project is to simplify debugging of a Viper program when using the Silicon backend. We approach this in two steps. First, we modify the Silicon backend such that the errors contain as much relevant information as possible. Then, we add new tools that enable the user to interact with this information.

Providing debugging information in a user-readable format. The most important step is to provide as much information as possible to the user. This includes the program state (e.g. variables, their (symbolic) values and information about the heap) and the path constraint. We will attach this information to the error object of each failed assertion and make it possible to display it in a user-readable format. This means that all information must be provided in a format that can be understood on the Viper level. Internally, Silicon uses a different representation that refers to functions, values and concepts that do not exist on the Viper level. Hence, we have to modify Silicon such that it also stores the Viper representation.

An issue will be that some constraints are a result of the Viper-internal encoding, such as properties of datastructures or artifacts of folding and unfolding predicates. Such constraints are most likely not of any help to the user and would just blow up the provided debugging information. Hence, hiding such artifacts would ease the analysis. An idea is to store the constraints in a hierarchical structure, such as a tree. For example, on the top level could be the constraint “unfold(p)”, the next lower level would then be (more or less) equal to the predicate definition. Recursively, there might be more levels.

Basic user interaction. Even with all the provided information the user likely has to experiment with additional assumptions and assertions. Instead of doing this by modifying the code, we want provide a tool that enables the user to interact directly with the symbolic state. The interactions include modifying the path constraint by adding and removing assumptions and modifying the to-be-proven assertion. As the symbolic state is attached to the error, verification does not have to be rerun. Instead the prover can be invoked directly, which will be faster and more convenient. This technique can also help to circumvent timeouts because the path constraint and assertion can be simplified in a way that speeds up the proof. Through these interactions the user gains valuable insights about the error and how to resolve the issue.

2.2 Possible Extensions

Once we have all the debugging information and the basic user interaction tool, there are a lot of extensions one can think of. These are not the main goals of the project but will be added if time allows for it. Otherwise they are left for future projects.

Choosing an SMT solver. Implications are proven by sending the path constraint and assertion to an SMT solver. By default Silicon uses the Z3 solver [3]. However, SMT solvers are not complete and hence, there might be statements that cannot be proven even though they hold. Every SMT solver works differently. It is possible that a statement can be proven by some solvers but not by others. Therefore, adding the possibility to choose an SMT solver (for example, Z3 [3] and CVC5 [1]) will be beneficial.

Proof rules. If a complex assertion cannot be proven, it might be useful to have a manual process to split it into smaller assertions, for example by splitting conjunctions and proving each part individually. Being able to apply such rules would be convenient to users.

References

[1] Haniel Barbosa et al. "CVC5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022.* Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Springer, 2022, pp. 415–442. DOI: `10.1007/978-3-030-99524-9\_24`.

[2] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (1976), pp. 385–394. DOI: `10.1145/360248.360252`.

[3] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Springer, 2008, pp. 337–340. DOI: `10.1007/978-3-540-78800-3\_24`.

[4] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. “Viper: A Verification Infrastructure for Permission-Based Reasoning”. In: *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Vol. 9583. Lecture Notes in Computer Science. Springer, 2016, pp. 41–62. DOI: `10.1007/978-3-662-49122-5\_2`.

[5] Malte Schwerhoff. “Advancing Automated, Permission-Based Program Verification Using Symbolic Execution”. PhD thesis. ETH Zurich, Zürich, Switzerland, 2016. DOI: `10.3929/ethz-a-010835519`.