Development of a data collection tool for the evaluation of a deductive verifier

Bachelor Thesis Project Description

Simon Hostettler
Supervised by Dionisios Spiliopoulos, Prof. Dr. Peter Müller
Department of Computer Science ETHZ

October 2023

Introduction

Deductive software verification is the usually automated process of proving the correctness of a program given a specification, based on a logical interference scheme. Viper [1] is a deductive verification infrastructure and an intermediate language to automate proofs using permission logics. This allows reasoning about heap-based and concurrent programs. The Viper infrastructure is based on two back-ends, one using verification condition generation through an encoding to Boogie [2], and one based on symbolic execution. Viper itself is human-readable and as such can be used to verify programs, but there is also a variety of front-ends based on the Viper infrastructure , such as Prusti [3] for Rust, Gobra [4] for Go, and Nagini [5] for Python.
Due to the nature of this tool-stack, Viper's performance can vary wildly for even small changes to the same program. Profiling this performance is difficult due to the variety of tools and levels of abstraction involved in the infrastructure. Especially for large projects like VerifiedSCION [6] reducing verification times is important.

Central Idea

To get a proper understanding of the sources of slow performance, a large data-set of different programs would be helpful. This could be used to profile the performance of the various stages of the Viper infrastructure in the future. Automatic program generation would be an option to create such a collection, however it might not represent a good sample of programs that are used in real word applications.

To create such a data-set, our goal for this thesis is to develop a data-collection tool for the Viper infrastructure. Whenever a program is verified by a user, either through one of the various front-ends or through Viper itself, the tool will automatically store

the program and associated metadata. This collection process will require user consent, through the use of an optional flag.

This tool will be made up of two core parts. The first part is the front-end in the Viper infrastructure itself, which will be invoked whenever a program is verified. We will use this to collect the programs and send them to the back-end, including any relevant associated information.
The second part will be a back-end. This will accommodate a database to store and evaluate the received data. To ensure the usefulness of the collected data, we will also want the back-end to filter the programs through certain criteria. We want to reduce duplicate or identical programs, by comparing the programs' structure and behaviour. We also want to filter out programs that won't prove useful to performance evaluation, such as ones containing syntax errors. Furthermore we'd like to keep the data varied in regard to certain features, such as the front-ends used for the programs.

Core Goals

Reduction of duplicate data

For this goal, we will develop and implement a similarity measurement to compare programs and filter out nearly identical ones. To achieve this, we will compare programs in regard to their structure, by comparing their source code. Furthermore, data from Viper such as the verification result or the chosen solving algorithms will also be taken into account, since programs that look similar but behave differently would be interesting to include in the data-set. Whenever a program is sent to the back-end, we will use this measurement to decide whether it should be kept or discarded.

Ensuring variety and usefulness of data

To achieve this objective, we will design and implement a way to analyse received programs to decide whether retaining them would improve the data-set. To do this we will first examine the potential utility of a program for performance measurements. We will, for example, filter out programs that fail to pass type-checking or contain syntax errors. We also want to keep the data-set balanced in regards to certain properties, such as front-ends used to compile the programs, language features used in the programs, and verification results for example. Programs featuring characteristics that are underrepresented in the data-set should be retained with a higher likelihood than ones that occur more frequently.

Feature selection

To improve the usefulness of the data for future evaluation, we want to store additional features and metadata alongside the programs in the data-set. For this goal, we will investigate which attributes might be necessary to include, and design our database accordingly. We will then include these features in the data that is sent back from

the front-end or generate them in the back-end, if they are independent of the original execution environment.

Back-end implementation

For this goal, we will write a program to set up the database and handle the receipt of the collected data. Here we will filter out any received program according to the criteria described in the first two core goals. Furthermore, we will evaluate the programs to measure their runtimes and any missing features.

Front-end implementation

For this goal, we will implement the collection tool in the Viper infrastructure. It will be invoked automatically whenever an opted-in user calls one of the Viper verifiers and send the program and relevant metadata to the back-end.

Extension Goals

- One extension goal will be to extend the tool to be able to evaluate changes made to the Viper source-code on the data-set. This will show performance-improvements or degradations in newer Viper versions.
- Another extension goal will be to artificially increase the size of the data-set. For this we will modify stored verifiable programs to introduce failures, and as such create new examples for the data-set.
- Another extension goal will be to extend the tool to make the data-set searchable for specific Viper code patterns.

References

[1] 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 - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016.

[2] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[3] Vytautas Astrauskas, Aurel Bílý, Jonás Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. The prusti project: Formal verification for rust. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan

Perez, editors, *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, pages 88–108. Springer, 2022.

[4] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. Gobra: Modular specification and verification of go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 367–379. Springer, 2021.

[5] Marco Eilers and Peter Müller. Nagini: A static verifier for python. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 596–603. Springer, 2018.

[6] VerifiedSCION. https://www.pm.inf.ethz.ch/research/verifiedscion.html. Accessed: 2023-09-22.