Visual Debugging for Symbolic Execution

Master's Thesis

16 September 2018

Alessio Aurecchia
aalessio@student.ethz.ch

supervised by
Arshavir Ter-Gabrielyan

Chair of Programming Methodology
Department of Computer Science
ETH Zürich

Abstract

In recent years, tools for program verification have made significant progress and are becoming more widely used. Yet, they still lack effective facilities to allow investigating and understanding verification errors, especially when the input program makes use of more advanced language features, such as quantified permissions in Viper.

We think the most effective way to help a user in identifying the source of a verification error is by employing a visual debugging approach, therefore we want to provide a technique to automatically produce small, visual counterexamples based on the information provided by a symbolic execution engine.

We conducted a feasibility study to understand whether we could effectively generate counterexamples via bounded modeling with Alloy, a language and analyser for software modeling. The main idea behind this methodology is that Alloy, because it performs a bounded search, is not affected by the problems involving quantifier instantiation, and is therefore able to provide complete concrete models in situations where the SMT solver would not.

In our technique, we encode the information about a symbolic execution state into a model and use Alloy to generate instances of it. These instances can then be used to build a visual diagram of the program's state. When the state being modeled is one where a verification error occurred, and we additionally encode the last failed query performed to the SMT solver, then the instances Alloy generates are counterexamples to the failed verification.

As part of our feasibility study, we have implemented a subset of the technique described in this thesis into a tool, integrated with the Viper IDE. This proof of concept for a debugger demonstrates that our approach for visualizing counterexamples to verification failures in the context of symbolic execution is a feasible one and is worth exploring in more detail.

Contents

1 Introduction 7
1.1 Motivation 7
1.2 Previous Work 8
1.3 Main Objective 9
1.3.1 Updating the existing infrastructure 9
1.3.2 Visualization of Quantified Permissions 9
1.4 The Structure of this Report 10
Bibliography **11**

1 Introduction

In recent years, tools for program verification have made significant progress and are becoming more widely used. The verification environments built around them provide features that help the user in writing valid specifications easily and, more importantly, they provide a way to work with the verifier in an interactive fashion. Yet, these tools still lack effective facilities to allow investigating and understanding verification errors, especially in the context of more advanced language features, such as quantified permissions[1], in the case of the Viper language[2].

1.1 Motivation

Determining the source of a verification error can still be tedious. The errors are found by the SMT solver[3], that "sits" at the lowest layer of most automated verification toolchains, and they are specified with respect to the SMT encoding of the program being verified. This makes it generally difficult to relate these problems back to the higher level of abstraction that the user works at. In addition to that, due to the inherent incompleteness of SMT solving, verification may fail for reasons other than the program being incorrect (e.g. for formulas involving quantifier or non-linear integer arithmetic), and the solvers may not be able to come up with models for more complex problems.

The IDE built around the Viper framework [2] has been largely expanded as part of Ruben Kälin's thesis [4] and allows fast feedback on the state and result of the proof when working with the verifier. Moreover, the project provides an early prototype for a Symbolic Execution debugger, laying down a solid foundation to build a more advanced debugger on. This debugger is limited in that it does not support visualizing structures defined via quantified permissions and its abilities of providing counterexamples to a verification error are lacking, mainly because of only visualizing symbolic states and having to rely on the SMT solver to provide the model for building a counterexample.

In the presence of advanced language features, this approach is not powerful enough to provide the user with effective visual representations of the problem, therefore we want to explore alternative ways of building counterexamples with the information

provided by the symbolic execution engine, and without necessarily having to rely on models from the SMT solver.

1.2 Previous Work

The original debugger offered support for navigating through the various states of the symbolic execution performed by Silicon, as well as inspecting their internal information. In addition to that, its graphical representation of the symbolic state supported visualizing local variables, functions, predicates and objects on the heap. These features were intended to provide a visual counterexample to a verification error, in order to help the user identify where the problem with the specification might be. This approach is limited by the fact that all the information about local variables or heap chunks is symbolic. For this reason, there is a limited amount of facts that can be learned directly about the values of local variables or whether, for example, references are aliases. The debugger tries to discover some of this information from explicit facts in the path conditions of the state being inspected.

The *VeriFast Program Verifier* [5] is a tool for the verification of C and Java programs annotated with preconditions and postconditions written in separation logic, and is implemented via symbolic execution, which make it, at least in principle, very similar to Silicon. The VeriFast IDE also provides debugging features similar to those provided by the original Viper Debugger: it allows navigating through the states explored by symbolic execution and displays the store, the heap, and the path conditions for each one of them, but does not provide a visual representation of this information.

The *Symbolic Execution Debugger (SED)* [6] is a platform for symbolic execution that allows debugging programs by exploring their symbolic execution tree interactively. In addition to that, SED also allows verifying programs (or parts of them) when JML[7] specifications are provided. The tool is implemented on top of the Eclipse IDE and uses KeY's Symbolic Execution Engine [8]. The system allows visualizing information about the current symbolic state being executed, such as the symbolic stack and path conditions. In case of potential aliasing between local variables, SED allows displaying graphs of the memory layout and provides a slider to inspect all the possible aliasing configurations.

Alloy [9, 10] is a language and an analyser for software modeling. It allows describing sets of structures via the use of constraints and then finding instances of these models, or counterexamples to assertions about facts in the model. The search space for these instances is limited to a *scope* defined by the user. The tool displays these structures graphically as an interactive graph. It is possible to "go to the next instance" so that all instances in the scope can be inspected manually. Moreover, Alloy performs symmetry-breaking to avoid showing multiple instances of the model that would essentially be equivalent. We are going to investigate whether Alloy can be used as the main tool to solve our problem of counterexample generation.

1.3 Main Objective

The overall objective we would like to achieve with this project is to find an effective way of visually representing dynamic structures defined via the use of quantified permissions. We will conduct a feasibility study in order to understand whether a bounded modeling approach can be employed to build these visualizations and to identify counterexamples when a verification error occurs.

1.3.1 Updating the existing infrastructure

The current version of the Viper IDE only provides features for writing Viper code, not for debugging it. The code in other parts of the framework has evolved and the compatibility with the debugging features of the IDE has been broken.

The first task of this project is to understand the requirements for the design of an infrastructure that would allow us to add debugging features to the IDE and then to actually put it in place. The architecture should allow extensibility and should not force other components of the system to depend on debugging features they do not need. The logging infrastructure of Silicon will also need to be updated in order to provide the debugger with all the information needed to visualise the verification states.

1.3.2 Visualization of Quantified Permissions

In order to achieve our main objective, we will devise a technique that will allow us to encode the information about a symbolic execution state into an Alloy model. The model will then be used to generate instances of the symbolic state

Our technique for modeling symbolic states will be integrated into a broader pipeline, that will allow solving some limitations of working with Alloy alone. In this thesis, though, we will only focus on the part of the pipeline that deals with modeling the program's state.

One important distinction between the approach of the previous debugger and the approach we take with the new one is that, in this new project, we focus on providing concrete counterexamples, whereas the visualization provided by the original debugger only showed symbolic states, where the value of variables were not known. This means that we also always display visualizations where the aliasing relation between reference is known.

Finally, we will conduct an evaluation to understand whether our technique is practically effective in visualizing counterexamples and whether it has fundamental limitations, both in terms of the approach and of its implementation.

1.4 The Structure of this Report

In this section, we give a short overview of the contents and the structure of this report.

Chapter 1 introduced the main motivation behind the project, it briefly described related work that has been done in the field, and it outlined the goals we would like to ultimately achieve.

In Chapter 2, we explain our approach to solving the problem of counterexample generation. We describe both the more general pipeline we envisioned and the subset that we focused on as part of this thesis.

Chapter 3 presents the translation of Silicon terms that appear in the symbolic execution trace into Alloy, and describes the technique we use to encode the symbolic state into a model that can be used to generate counterexamples.

In Chapter 4 we discuss the previous architecture of the system and how it was updated to enable integrating the new debugger extension. In chapter 5 we briefly describe the features offered by the debugger.

In Chapter 6 we evaluate the models produced by the debugger by considering some concrete use cases and we discuss its limitations in terms of the technique and its implementation. Then, we compare our approach with that of other program verifiers based on symbolic execution that offer debugging features.

Finally, in Chapter 7 we draw conclusions about the project and briefly list some potential interesting topics for future work.

Bibliography

[1] P. Müller, M. Schwerhoff, and A. J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification (CAV)*, volume 9779 of *LNCS*, pages 405–425. Springer-Verlag, 2016.

[2] 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.

[3] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Satisfiability modulo theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. 1 edition, 2009. ISBN 9781586039295. doi: 10.3233/978-1-58603-929-5-825.

[4] Ruben Kälin. Advanced features for an integrated development environment. Master's thesis, ETH Zürich, 2015.

[5] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 41–55, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-20398-5.

[6] Martin Hentschel, Richard Bubel, and Reiner Hähnle. The symbolic execution debugger (sed): a platform for interactive symbolic execution, debugging, verification and more. *International Journal on Software Tools for Technology Transfer*, Mar 2018. ISSN 1433-2787. doi: 10.1007/s10009-018-0490-9. URL `https://doi.org/10.1007/s10009-018-0490-9`.

[7] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with jml and esc/java2. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 342–363, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-36750-5.

[8] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*.

Springer, 2016. ISBN 978-3-319-49811-9. doi: 10.1007/978-3-319-49812-6. URL http://dx.doi.org/10.1007/978-3-319-49812-6.

[9] AlloyTools. alloytools.org, 2017. URL http://alloytools.org/.

[10] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012. ISBN 0262017156, 9780262017152.