Advanced Features for an Integrated Verification Environment

Master Thesis Description

Ruben Kälin
rukaelin@student.ethz.ch

May 3, 2016

Introduction

Tools for verifying program correctness are becoming more and more powerful and widely used. However, it is still difficult for a non-expert in program verification to debug verification errors. Current verification environments provide helpful support for proving properties of software, such as syntax highlighting, auto-completion, and visualization of syntax errors, but lack assistance for fixing errors in proofs. In case of a verification failure, the shown warnings and error messages are often cryptic due to being generated from the Satisfiability Modulo Theories (SMT) solver at the bottom of the program verification toolchain.

Moreover, it is often already difficult to distinguish spurious errors, e.g. originating from an incompleteness of the SMT solver, from problems in the specification or in the code to be verified.

For a spurious error the specification might need to be reformulated in order to enable a successful proof. Another possible reason for the verification to fail is a timeout, in which case simply increasing the timeout might solve the issue. If the problem is due to a wrong specification, it is unclear where to apply the fixes, because the error message often only points to the assertion it failed to proof. In case the SMT solver found a counter-example, verification experts can track down the source of the problem in a colloquial and manual process. An automation of this process is desirable to improve the productivity of verification experts and enable software developers to find causes of the errors.

In the current project we address the problem of simplifying the analysis of failed verification attempts by building a debugger for symbolic execution in the context of the Viper program verification infrastructure, [Viper]. On one hand, the tool should enable software developers to debug verification errors they were previously not able to track down. On the other hand, we want to increase the productivity of verification experts.

The project can be structured in three parts as shown in Figure 1. Automation deals with inference relieving the programmer from manually having to do simple tasks. The use of external tools, e.g., the axiom profiler, is part of the integration. The feedback part is concerned with showing the necessary information to the programmer in order to assist her in finding the reasons for failed verification attempts.

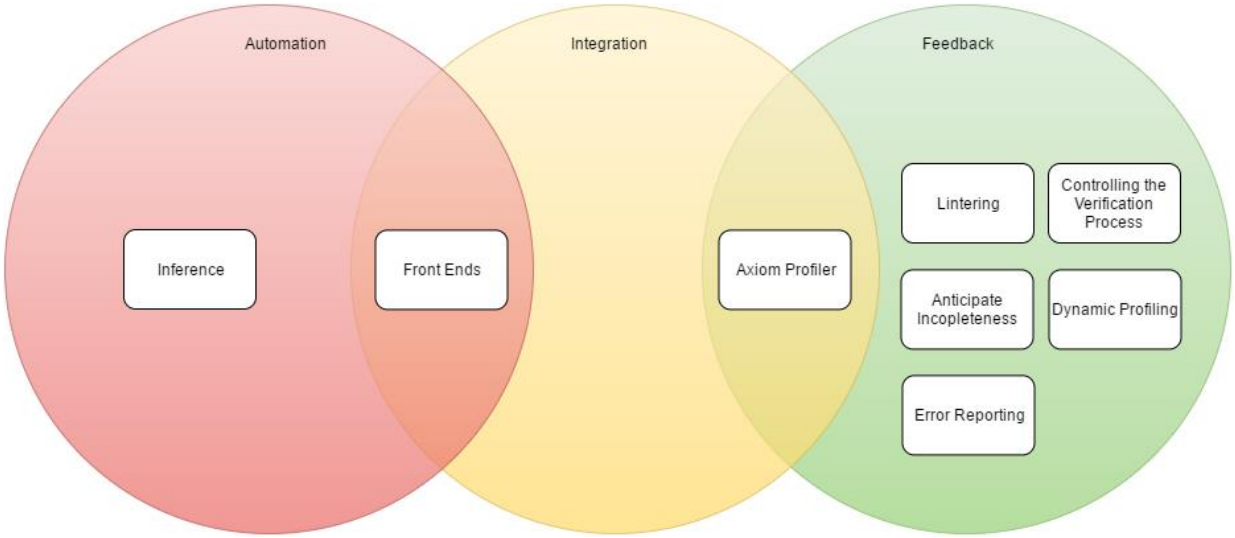

Figure 1. Project Overview

Previous work

In his master thesis, Ivo Colombo envisioned a debugging tool for Symbolic Execution (SE), [Ivo]. His thesis contains many helpful ideas about visualizing the state of an active debugging session in order to convey the information in an easily accessible and orderly fashion without cluttering the view. In the current project we plan to justify the design decisions regarding the visual representations of the debugging features by a comparison to [Ivo].

Most existing work is focused on including debugging functionality into a Verification Condition Generation (VCG) based toolchain, e.g., [Dafny] and [Boogie]. However, even though VCG and SE are fundamentally different ideas, we can still draw inspiration from those projects, because both approaches are based on an SMT solver. We plan to guide some of the visual aspects of this project on the way the Dafny debugger works by providing a similar functionality for the Viper toolchain.

Currently there exists a lightweight IDE solution based on the text editor Sublime Text 3. This IDE can be used to edit Viper intermediate language source code and provides syntax highlighting and syntax checking. It allows verifying programs using either the VCG backend or the SE backend from the IDE. As a result of the verification, the IDE displays the time spent verifying and either a success or an error message. In case of a verification error, possible assertion violations are marked in the source code.

Infrastructure

The Viper infrastructure consists of two verification back ends: A Verification Condition Generation (VCG) backend and a backend that proves the correctness of programs using Symbolic Execution (SE). Because there already exist tools for debugging VCG, we choose to implement debugging support for SE. We plan to build the debugger in a way that allows adding the support for VCG in the future.

The debugging system for SE is most useful when integrated into an IDE. We would like to be able to write, verify, and debug the code to be verified in one tool. We plan to incorporate the functionality as an extension into the VS Code text editor. The IDE should support the same basic functionality as the previous, Sublime based, system including syntax highlighting, syntax checking, and a mechanism to start a verification using either the VCG or the SE based backend.

We found Microsoft's Visual Studio Code (VS Code) to be the text editor best suited for our purposes, [VSCode]. This extensible open source project shows better performance than its contestant Atom, [Atom]. Moreover, the superior structure of VS Code allows it to be enhanced either by implementing an extension using VS Code's extensibility API or modifying the API itself. Most of the

graphical features needed for this project are supported by VS Code, and the possibility of extending the API allows for further customization.

Feature Descriptions

In this section we list all features relevant for an integrated verification environment (IVE). This project will only tackle a subset of those tasks. In section Prioritization of Features, we discuss the importance of each task for this project and cluster them into core tasks and extensions.

The features in this section are divided into four categories. Each category represents a phase of the verification process. In the pre-verification, category 1, the programmer is assisted in editing the source code and its specifications. Category 2 talks about Information reporting, which takes place when the user requires semantic information. E.g., when the information reporting is started, inferred properties such as triggers and ghost operations are shown. Category 3 is concerned with control and feedback of a running verification. After a failed verification, the IVE reports the detected errors and assists the process of finding and fixing the cause of the problem. For example, the state of a counter-example, i.e., an execution violating the specification, is presented to, and can be explored by the user. Figure 2 shows a possible interaction with the IVE.

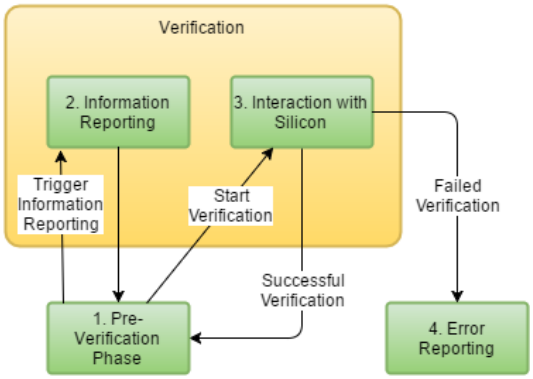

Figure 2. Diagram of the verification phases

1. Pre-Verification Phase

Before the verification is started the programmer is concerned with editing the code. Therefore, a good IVE provides support for editing such as code completion and dynamic syntax checking. To help the programmer in annotating a program with contracts, some of the specifications can be generated using inference techniques.

Concretely, we consider the following features:

- Code completion and syntax highlighting
- Dynamic syntax checking
- Specification inference

Code Completion and Syntax Highlighting

The predecessor of the proposed system already provided basic assistance, such as syntax highlighting and code completion. We plan to reimplementation this functionality for the new IVE.

The existing verification support is using a different parser than the Viper compiler. Ideally we could use the same incremental parser for the new IVE as for Viper.

Dynamic Syntax Checking

Parsing errors should be shown directly in the code by underlining the erroneous parts of the program. A tooltip can convey more information explaining the problem or suggesting a possible fix.

Specification Inference

There already exists a separate tool for specification inference for Viper, [Sample]. Including this specification inference into the IVE would improve the programming experience.

2. Information Reporting

Information Reporting happens after phase 1 of verification, because it depends on the output of the verification preprocessing. Since it is computationally intensive to verify a program, information reporting can save time. The displayed inferred triggers can be checked by the programmer. Any detected incompleteness issues can be reported without the need to run verifier.

Here, we list the features relevant for information reporting:

- Incompleteness anticipation
- Trigger inference
- Inference of ghost operations

Incompleteness anticipation

Due to the undecidability of the underlying proof problems the SMT solvers are incomplete. Anticipating this incompleteness and informing the programmer about detected issues in the form of warnings facilitates the verification.

Trigger Inference

In order to verify programs the SMT solver needs to be provided with triggers. Coming up with the right triggers is difficult, [VCG]. However, since Triggers are always inferred, unless explicitly specified, displaying them allows programmers to check their appropriateness. We hope that showing the inferred triggers in the IVE improves usability, even though the programmer still needs to assess each of them.

Inference of Ghost Operations

Ghost operations facilitate the verification. Those modifications on the written source code are generated automatically, in the verification preprocessing step, and are thus invisible to the programmer. In order for her to learn about the ghost operations, we should make them visible in a tooltip upon request.

3. Interaction with Silicon

During the verification some gathered information could be shown to the programmer. For example the progress can be shown as a progress bar.

The considered features are:

- Progress reporting
- Controlling the verification process

Progress Reporting

Currently the Viper system is only informing the programmer about the outcome of the verification once it is finished. Therefore, the programmer cannot decide whether a slow verification only takes longer than expected or whether it does not terminate at all, e.g. due to a matching loop, [VCG]. Adding a visual progress bar to the IVE displaying the progress of the verification could alleviate this problem. The Z3 SMT solver provides feedback about running verifications that can be used to implement such functionality, [Z3].

Controlling the Verification Process

The programmer should be able to start a verification using either the VCG or the SE backend. Moreover, it should be possible to abort a running verification from within the IVE. Upon completion, the verification should display a success message or mark the detected problems in the source code.

4. Error Reporting

After a failed verification the IVE should help to pinpoint the source of the problem. The error reporting system should be able to show the information relevant for the unsuccessful verification. The user should also be able to debug the verification by executing the actions one by one.

Here, we list the most important features of the error reporting phase:

- Counter-example visualization
- Stepwise debugging
- Triggering traces
- Verification path visualization
- Heap visualization
- Integrated static profiling
    - Axiom Profiler
    - Profiling Symbolic Execution

Counter-Example Visualization

The debugging system should import the counter-example provided by the SMT solver (Z3), [Z3]. These counter-examples can then be shown directly in the code, [Z3]. Therefore, we map all states of the counter-example to a code location that can be highlighted with a marker. Once a state is selected the values of all variables relevant to finding the problem can be displayed in a separate frame in the IVE. The frame could look similarly to the one developed for Dafny, [Dafny, Figure 4].

Stepwise Debugging

For an unsuccessful verification it is interesting to step through the code. For each step, the associated verification actions can be shown in a separate frame and executed one by one. It should be possible to step forward as well as backwards through the code.

Triggering Traces

In order to use a quantifier, Viper needs to instantiate its body with concrete terms that match the quantifier's trigger. Displaying the triggers that have been instantiated during verification can be helpful. E.g., determining the reasons for non-termination or very slow verification is facilitated by knowing the instantiation graph.

Verification path visualization

During a debugging session the path that was taken should be visually displayed in the source code. E.g., in an if-then-else statement, the branch that is taken should be marked, and the one that is skipped can be grayed out as described in [Ivo, Figure 3.10]. The code with the visualization should make it clear what execution path has been taken.

For example, in XCode, the execution paths are visually shown in the code when analyzing for memory leaks. Figure 3 demonstrates the execution path visualization in XCode, [MemLeak]. The arrows mark the control flow directly in the source code. Such a visual feature would not be available for VS Code through the native API. It could only be implemented by adding it to the extensibility API of VS Code.

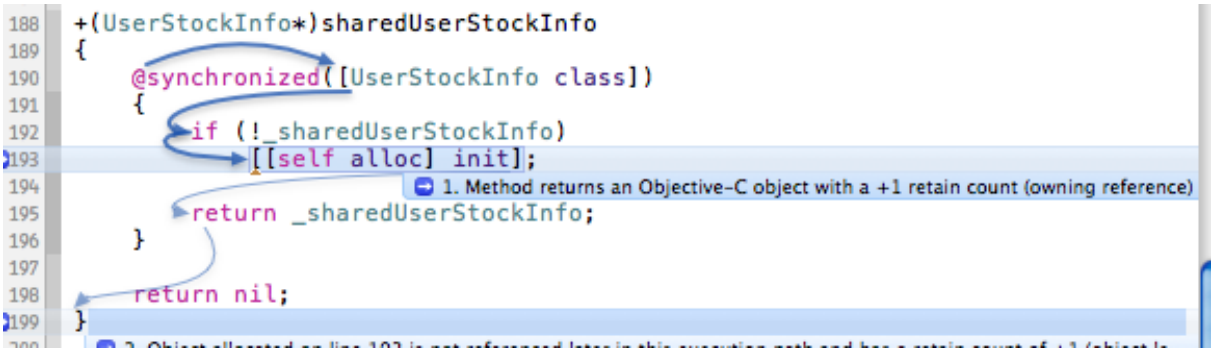

Figure 3. Visualization of an execution path in XCode.

Heap Visualization

Program verifications often reason about the state of the heap. Therefore, it is very helpful for debugging to have a visualization of the heap at certain points of execution. The heap can be represented as a graph where the nodes are objects and the links are pointers. For example, such visualization for heaps has been envisioned in [Ivo, 3.4.6.2].

Integrated Static Profiling

Microsoft Research provides an Axiom profiler for Z3 that could potentially be integrated into the IVE [Z3]. We do not plan to do this, because the profiler is mostly used by the developers of verification systems, whereas the primary focus of our project is on the needs of software developers. The Axiom profiler might be integrated in a future work.

Currently there is no tool for profiling symbolic execution directly. Building such a tool would be valuable and could be the subject of another project.

Long-term Tasks

In this section we include all tasks that are of importance for the IVE in general, but are not part of the current project.

Debugging VCG Back Ends

The debugging system for symbolic execution should be designed to allow an extension to also support VCG sub-system. Implementing this extension can then be done by supplying the needed information to the debugging system, similarly to the support for the symbolic execution. However, the implementation of this extension is out of scope for the current project.

Integration with Front Ends

The information gathered and shown by the IVE relates to the Viper intermediate source code. However, since programs can be translated from languages such as Chalice, Scala, or Python to the Viper intermediate language, it is interesting to think about mapping the debug information back to the source language. Since the current state of the existing Viper front ends is incipient, integration is out of scope of the current project. We none the less mention it here for completeness.

Prioritization of Features

In this section we propose an ordering of the features by their priorities and classify the tasks as core or extension tasks.

Core Tasks

As core tasks, the following features should be embedded into the IVE.

1. Code completion and syntax highlighting
2. Dynamic syntax checking
3. Controlling the verification process
4. Counter-example visualization
5. Verification path visualization
6. Progress reporting
7. Stepwise debugging

Extensions

The following list contains the tasks we propose as extensions.

8. Heap visualization
9. Inference of ghost operations
10. Specification inference
11. Trigger inference
12. Triggering traces
13. Anticipate incompleteness

Future Work

- Integrated static profiling
- Debugging VCG back ends
- Integration with front ends

Schedule

The project is approximately structured as shown in the following time schedule. It takes 6 months in total, starting on May 3, 2016.

The time distribution plan is below:

Task	**Time**	**Start Date**
Study related work, familiarize with the technical environment	2 weeks	May 3, 2016
Reimplement the functionality of the predecessor into the new IVE. (Tasks 1,2,3)	4 weeks	May 17, 2016
Implement counter-example visualization (Task 4)	2 weeks	June 14, 2016
Implement verification path visualization (Task 5)	2 weeks	June 28, 2016
Implement progress reporting (Task 6)	1 weeks	July 12, 2016
Implement stepwise debugging (Task 7)	2 weeks	July 19, 2016
Work on extensions	8 weeks	August 2, 2016
Write the report	4 weeks	September 27, 2016
Project Deadline		November 3, 2016
Final Presentation		November 10, 2016

Bibliography

[Viper] P. Müller, M. Schwerhoff, and A. J. Summers (2016): Viper: A Verification Infrastructure for Permission-Based Reasoning. Available at http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=MuellerSchwerhoffSummers16.pdf

[Dafny] K. R. M. Leino and V. Wüstholz. The Dafny integrated development environment. In Formal-IDE, volume 149 of Electronic Proceedings in Theoretical Computer Science, pages 3-15. Open Publishing Association, 2014. Available at http://arxiv.org/pdf/1404.6602.pdf

[Boogie] Claire Le Goues, K. Rustan M. Leino & Michał Moskal (2011): The Boogie Verification Debugger (Tool Paper). In Gilles Barthe, Alberto Pardo & Gerardo Schneider, editors: Software Engineering and Formal Methods (SEFM), LNCS 7041, Springer, pp. 407–414. Available at http://dx.doi.org/10.1007/978-3-642-24690-6_28

[Ivo] I. Colombo, M. Scherwoff, P. Müller (2012): Debugging Symbolic Execution. Available at http://www.pm.inf.ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Ivo_Colombo_MA_report.pdf

[Z3] L. Moura, N. BjørnerZ3 (2008): An Efficient SMT Solver. Available at http://link.springer.com/chapter/10.1007/978-3-540-78800-3_24

[Sample] P. Ferrara and P. Müller: Automatic inference of access permissions Verification, Model Checking, and Abstract Interpretation (VMCAI), 2012. Available at http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=FerraraMueller12.pdf

[MemLeak] Available at http://www.asksat.com/questions/639359/objective-c-memory-leak-issue-in-class-method

[VCG] S. Heule, I. T. Kassios, P. Müller, A. J. Summers: Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions, 2013. Available at http://people.inf.ethz.ch/summersa/wiki/lib/exe/fetch.php?media=papers:predicates.pdf (P. Müller, 2016)

[VSCode] Available at https://code.visualstudio.com/

[Atom] Available at https://atom.io/