Creating an Advanced Debugger for Symbolic Execution

Master's project description

Alessio Aurecchia

supervised by Arshavir Ter-Gabrielyan

March 2018

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.

Determining the source of a verification error can still be tedious. The errors are found by the SMT[1] solver that "sits" at the lowest layer of most 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, the matter is made even more complex by the inherent incompleteness of SMT solving, which means verification may fail for reasons other than the program being incorrect.

The IDE built around the Viper framework [1] has been largely expanded as part of Ruben Kälin's thesis [2] 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 Semantic Execution debugger, proving that it is, in fact, possible to build such a tool and laying down a solid foundation to build a more advanced debugger on.

The IDE offers three levels of tool support for working with the Viper framework: writing Viper code, verifying it, and debugging failed program proofs. With this project, we focus on the third of these levels in order to provide a tighter integration with the external tool by enabling the user to **interactively** investigate verification failures at the level of abstraction of the Viper language, and not of its implementation. Our main objective is to update the Viper IDE so that

[1] `https://en.wikipedia.org/wiki/Satisfiability_modulo_theories`

the existing debugger prototype works with the current version of the Silicon backend (see Core Goals). After that, we will build on top of it and experiment with new ideas on how to debug and visualise Viper programs written using quantified permissions (discussed in section Extensions).

Previous Work

The existing Viper IDE provides a series of features to help users write Viper code (e.g., syntax highlighting, code completion, automatic syntax checking) and also offers a debugger which allows visualising and exploring the symbolic heap in different states of a failed verification attempt.

The IDE has proven to be a very useful tool for quickly verifying programs with the Viper framework and is used for a large part of the Program Verification course [2]. In addition to that, the built-in debugger prototype provides a solid foundation for building an advanced debugger for symbolic execution.

The Viper IDE provides two debugging modes: the *simplified debugging mode* and the *advanced debugging mode*. The simplified mode allows the user to explore a restricted set of verification paths that lead to a verification failure (called *reference states*) and compare any of them with the error state. The advanced mode allows the user to inspect and compare any pair of states, not only those that are on the path to a verification failure. This second debugging mode is aimed more at developers of the Silicon backend rather than users of the verifier. In both debugging modes, the IDE displays markers on the source code that denote verification states related to the currently selected one (non-relevant ones are hidden). Clicking on these markers changes the selection of states to display.

Working with recursive predicates can become cumbersome if the way data is accessed does not follow the way the structure of the predicate, that is why *Quantified Permissions* were introduced to the Viper framework [3]. These enable the users to avoid specifying many manual proof steps when working with structures such as arrays, cyclic data structures or graphs. Despite their usefulness, structures on the heap defined via the use of quantified permissions cannot currently be visualized by the Viper debugger, therefore adding support for them is one of the goals of this project (see Extensions).

In his Master's Thesis, Ivo Colombo worked on building a debugger for the Chalice language on top of the Syxc verifier. Despite the lack of more advanced language features (such as quantified permissions) in Chalice, Ivo's work provides some useful insights and guidelines on building a debugger for a symbolic execution verifier (in fact, Ruben Kälin's work is also based on some of these observations). The ideas that are most relevant for us are those regarding the conceptual design of a symbolic execution debugger [4, sec. 3]

[2] `http://www.pm.inf.ethz.ch/education/courses/program-verification.html`

The *VeriFast Program Verifier* [5] is also implemented via symbolic execution and provides a visual debugger that allows exploring the states of verification. This project can be studied to understand which ideas are effective and which are not for debugging symbolic execution.

Alloy [6, 7] is a language and analyser for software modelling. It allows describing sets of structures via the use of constraints and then finding instances (or counterexamples) of these models. 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.

The *Symbolic Execution Debugger (SED)* [8] is a platform for symbolic execution and allows to interactively debug programs based on symbolic execution. In addition to that, SED also allows verifying programs (or parts of them) when JML [3] specifications are provided. SED is implemented on top of the Eclipse IDE and uses KeY's Symbolic Execution Engine [9]. Debugging is performed by exploring the symbolic execution tree of the program. The system allows visualizing information about each state such as the symbolic stack, path conditions, and the memory layout. In case of potential aliasing, SED provides a slider to change the visualization of the memory layout between all possible configurations.

Core Goals

The overall goal 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.

Updating the existing infrastructure The current version of Viper IDE currently 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 the project is to gather the requirements for the design of an infrastructure that would allow us to add debugging features to the IDE and then to actually put in place that infrastructure. 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.

Visualization of Quantified Permissions In order to achieve our main objective we will first define some significant examples of Viper programs written using quantified permissions that are going to be used as the main use cases of our debugger. After that, we will have to experiment and design the most suitable visualization for those use cases and finally we will implement basic support for visualising and debugging them.

With the basic support for quantified permissions (the fall-back solution), we aim at providing a simple visualization for some specific debugging use cases. These visualizations will not necessarily be 100% usable (for example, they may be too big to work with), but they will still

[3] `http://www.eecs.ucf.edu/~leavens/JML/index.shtml`

provide a complete view of the structures on the heap. As part of the extensional goals, we want to devise the best possible way to simplify and to layout this information, in order to provide the most usable visualization to the user, assuming all the information needed to do that is already available. In the Schedule we refer to this task as *"the best" QP support.*

Extensions

Understanding heap structures

In order to visualize dynamic structures defined via quantified permissions, we need to understand what type of structure a group of locations on the heap actually represents. Initially, we will work with the assumption that we have all the information needed to visualize the structures on the heap, then work on ways to gather it from different sources:

1. Verifier / Encoding The most immediate way for the debugger to understand the structure of the heap is to extract the information from the Viper program itself. It should be possible to match certain encoding patterns to some types of visualization, by using heuristics. For example, specifying access to locations via ordered integers is likely to be used when denoting an array-like structure and a suitable approach at visualizing that information would be to display the memory locations being accessed as consecutive on the heap diagram, possibly with an indication of the lenght of the array with respect of variables in the store.

Moreover, in some situations, it is possible that we have complete information about aliasing from the encoding itself. E.g., in the snippet

```
if (x != y) {
  // s1
} else {
  // s2
}
```

Silicon definitely knows that in `s1` there is no aliasing between `x` and `y`, whereas in `s2` the variables are aliased.

2. Additional Queries to the SMT Solver In case not all of the information needed for visualizing the heap can be gathered from the verifier, we have the option of sending some additional queries to the SMT Solver directly, bypassing the verifier. The debugger may try to perform some queries in order to understand whether the user specification of the heap is equivalent to known model specifications. For example, we may be able to prove that the requirement

```
requires len(a) == |s|
requires forall i: Int :: 0 <= i && i <= |s| ==> i in s
requires forall i: Int :: i in s ==> acc(loc(a, i).val)
```

is in fact equivalent to

```
requires forall i: Int :: 0 <= i && i < len(a) ==> acc(loc(a, i).val)
```

3. User guidance The last resource for determining what heap locations represent is direct user guidance. Ideally, this is only needed in ambiguous situations and should be reduced to the minimum (if possible it should be avoided completely). Initially we will assume that we can have all of the guidance we want and then see how much of that information can be inferred via heuristics.

In case guidance is needed, the debugger could allow a way for the user to specify her assumptions about the data so that the heap locations can be visualized according to them. Moreover, the user could get immediate feedback in case the specification is incompatible with her assumptions.

Drawing useful models of the state

Aliasing One additional problem we have to consider when visualizing the heap is that of *aliasing* between references. Aliasing describes the situation in which different variables refer to the same memory location on the heap. Aliased references would be represented with separate arrows pointing to the same container in the heap visualization, whereas non-aliased references would point to separate containers.

Displaying references becomes problematic when we do not have full information about possible aliasing between references. Drawing two arrows pointing to the same container may be confusing for the programmer since it is not certain that the references are aliases of each other. On the other hand, pointing the arrows to separate containers would also be ambiguous, as it is not necessarily true that the two references are distinct (and we might mislead the user into thinking that they **cannot** be aliases).

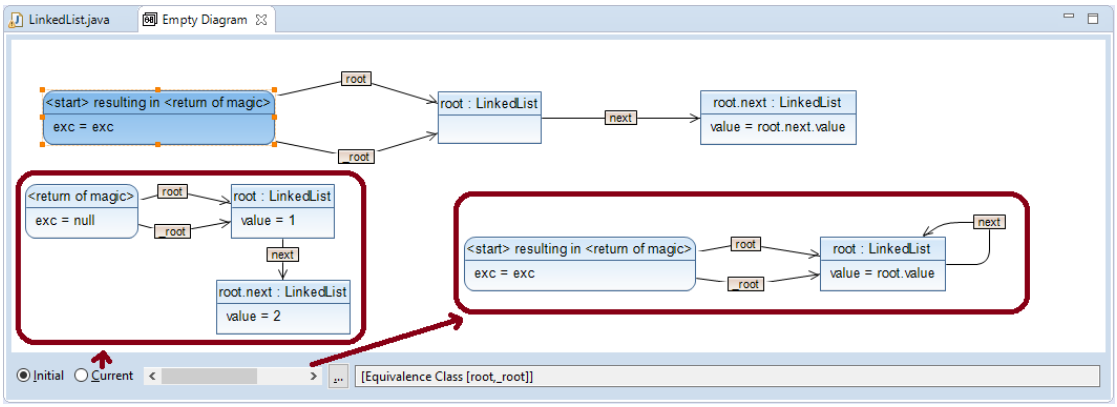

Figure 1: Inspecting the memory layouts caused by aliasing in SED. The scrollbar changes which of the two visualizations in the red boxes is shown. Only one concrete situation is visible at any time. [4]

Currently, the Viper debugger draws references pointing to different boxes on the heap, despite some of them possibly being aliases. In SED (see Previous Work) the memory layout visualization provides a way of choosing different visualizations in case there might be aliasing, as displayed in figure 1 .

Sub-heaps Another issue we have to consider is that of "*sub-heaps*", the situation in which one or more structures are part of a larger structure. Again, we have the problem of how to detect this situation and also of how to visualize it.

The problem arises when dealing with locations that we know are part of a larger data structure (for example sections of an array or list) since we previously had that information, but we cannot come to that conclusion because the verifier had to "forget" some of that information in order to proceed with the proof.

Schedule

The following table outlines the estimated time for each task during the course of the project and the corresponding starting date.

[4]Picture from `http://i12www.ira.uka.de/key/eclipse/SED/index.html`

	Task	Time	Start Date
C	Learning phase: getting to know the framework's infrastructure and previous work / Finding example use cases	2 weeks	26.03.2018
C	Gather requirements for the design of the infrastructure	1 week	09.04.2018
	Prepare Initial presentation (19.04.2018)	0.5 weeks	16.04.2018
C	Enable the new infrastructure design	5 weeks	19.04.2018
C	Design QP visualization (for the use cases found)	2 weeks	24.05.2018
	Prepare Intermediate presentation: Design (14.06.2018)	1 week	07.06.2018
C	Implement basic QP support (Fall-back)	2 weeks	14.06.2018
E	Implement "the best" QP support (assuming all needed information is available)	3 weeks	28.06.2018
E	Implement the heuristics to gather the information	3 weeks	19.07.2018
E	Re-enable Ruben's debugging features on top of the new architecture	1 weeks	09.08.2018
E	More Testing, Continuous Integration (Jenkins), Additional features	1 weeks	16.08.2018
E	Documentation	1 weeks	23.08.2018
	Writing the thesis	2.5 weeks	30.08.2018
	Project Deadline		16.09.2018
	Final Presentation		24.09.2018

C: Core Goal E: Extensional Goal

References

[1] 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.

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

[3] 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.

[4] Ivo Colombo. Debugging symbolic execution. Master's thesis, ETH Zürich, 2012.

[5] Bart Jacobs and Frank Piessens. The verifast program verifier. Technical report, 2008.

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

[7] Daniel Jackson. *Software Abstractions.* MIT Press, 2012.

[8] 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.

[9] 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.