SMT models for verification debugging

Master Thesis' Project Description

Cédric Stoll

supervised by Dr. Alexander Summers and Arshavir Ter-Gabrielyan
Department of Computer Science, ETH Zürich

06.11.2018

Introduction

Trying to formally verify software code can be a cumbersome task even with help of highly automated tools, e.g. Viper [1]. A failed verification attempt leaves the developer with an "Assertion might not hold" exception and no further help in identifying the possible errors he has made.

A debugger that could give some hints, e.g. counterexamples to his current proof attempt, would help enormously. Alessio Aurecchia's master thesis "Visual Debugging for Symbolic Execution" [2] tackled this problem. For a subset of the Viper language it can generate small counterexamples and displays them visually in the IDE. The software modelling tool Alloy [3] was used to generate possible counterexamples. Using the symbolic execution log, generated by one of the Viper verifiers (Silicon), they generated an Alloy program for a failed assertion. The model for this Alloy program then corresponded to a possible counterexample for the original program. This approach has a few drawbacks: Alloy does not natively support theories and it has to bound the range of integers it considers. This led to approximations, which in consequence resulted in spurious counterexamples.

In this thesis, we will try an alternative approach to generate counterexamples. Silicon uses an SMT solver, Z3 to be specific, to verify the code and specifications. If a verification step fails, we want to use Z3 to produce a counterexample to the current failing assertion. We need to model the constraints given by the program in SMT and bound to the number of objects on the heap. We need to bound the size of the example, because otherwise we cannot display it visually in the IDE. After we get a model from Z3, we have to translate it back to a counterexample of the program. The advantage of using Z3 over Alloy comes in the native theory support that Z3 has. This theory support will allow us to model the theory constraints precisely and we will not get spurious models based on an incomplete theory support.

As the constraints inside the Viper program can contain a lot of quantifiers and some undecidable theories, Z3 could output "unknown". This means the solver could not satisfy the formula, but it also does not know if the formula is unsatisfiable. In the "unknown" case, we only get a partial model without any relevant information that is needed to construct a meaningful counterexample. It will be part of the project to tackle this problem and check if it is necessary to change the constraints. If we have to weaken the constraints, we could get some spurious counterexamples. However, e.g. replacing quantifiers with the instantiations made at the point of failure might eliminate such problems.

As our approach only alters the generation of counterexamples, we want to support the rest of the Silicon debugging pipeline. But there are also some further improvements we could make to the pipeline. We could include the feature for finding inputs which led to a failing state. Or more generally, find corresponding program states for a statement that precedes a failing assertion. Additionally we could try to support the other Viper verifier (Carbon); to achieve this goal we would need to abstract away from the symbolic execution log (as only Silicon generates it) and find a more general way to generate the counterexamples.

Core Goals

Build an example set

We need a set of examples of different verification failures. The examples should represent typical cases we find while debugging verification problems, e.g. verification errors due to weak specifications or faulty code. But also it should contain some cases where Z3 failed to find a model due to incompleteness.

We can take some from the previous thesis [2] and from the Viper project, but we also have to build more on our own. The difficulties in this example set should range from simple (no theories/quantifiers involved) to difficult (non-linear integer arithmetic and quantifiers).

Modelling in SMT

We need to model the program state in SMT. For this task we could use the constraints generated by Silicon or build a new system. In either way, we need to be able to remove quantified constraints. As in the evaluation phase, we want to test Z3 with queries without those special constraints.

Using Z3 as a model finder

Given the generated SMT formula we can evaluate Z3 as a model finder. We want to build an iterative algorithm to find a small enough counterexample. Therefore the SMT formula generated needs to bound the amount of objects on the heap. As this should be efficient, we need to find a way to reuse the calculations done for smaller sizes.

Alternative: using Z3 as an oracle

If our approach of using Z3 as a model finder fails, we could still use our modelling to SMT to verify the models found by Alloy. To do this, we would generate the SMT formula for the current program and then assume the counterexample found by Alloy. The assertion should still fail in this setting. Further, if Alloy outputs only a partial model,e.g. because of approximations of theory information, we could use Z3 to find additional information and complete the counterexample.

Implementation

In the end, we want to implement one of the two alternatives above. It should support the given pipeline for Viper and use the counterexample visualization system developed in [2].

Extensions

Matching program states

Finding a counterexample for the state of a failing assertion is helpful, but it does not show the programmer how the program reached this state. It would be helpful if one could select a preceding statement of the failing assertion, for which we already calculated a counterexample, and then it would generate the specific program state at this statement. It must hold that if the program was in that generated state, then the assertion under consideration will fail (and reach the state of the counterexample). This technique could be used to generate inputs for a method that will lead to an assertion failure.

User-provided constraints

We want to support user-provided constraints, which filter counterexamples generated by the debugger. Currently this could be done by additional assume statements or preconditions. An advantage of this feature would be a clear separation of debugging constraints (the new constraints to filter counterexamples) and the assume statements and preconditions specified to reason about the code.

Support for Carbon

The current debugger pipeline and the pipeline planned for this thesis work with the symbolic execution log generated by Silicon. To support Carbon we have to abstract away this source of information into a more general language that supports both verifiers.

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] Alessio Aurecchia. Visual debugging for symbolic execution. Master's thesis, ETH Zurich, 2018.

[3] Alloy. alloytools.org. `http://alloytools.org/`. [Online; accessed 24-October-2018].