Debugging Symbolic Execution

Master Thesis Description, March 2012
***Ivo Colombo**, icolombo@student.ethz.ch*

Supervisors: Malte Schwerhoff, Prof. Dr. Peter Müller

Introduction

Symbolic execution [1] can be used as an approach to automated software verification where code is evaluated with symbolic instead of concrete values. The computed symbolic values can be proven (e.g. using a theorem prover) to satisfy a formal specification of the program in the form of invariants, pre- and postconditions. If a proof fails, this can have various reasons, e.g. a wrong implementation and wrong specifications or underspecifications. It is usually left to the programmer to find the problem using the data reported by the verifier.

Chalice [2] is a research language for concurrent programs. It includes the implicit dynamic frames [3] methodology by using a permission model to restrict read and write access to heap locations, thus defining an upper bound on the set of heap locations a method depends on.

Syxc [4] is an automatic verifier for a subset of the Chalice language that uses the approach of symbolic execution. In case of failure, Syxc reports the location and type of the failure. It is also able to output the symbolic state at every location in the program that was reached during the verification.

The symbolic state of Syxc $\sigma = (\gamma, h, g, \pi)$ is composed of a symbolic store γ of local variables (mapped to their symbolic values), two symbolic heaps h and g (current and pre-state), both mapping heap chunks to their symbolic values, and a set of path conditions π which denote knowledge about symbolic values and their relationships in the form of first-order logic formulae, assumed during the symbolic execution of a program.

The symbolic execution approach together with the above-mentioned framing technique allows for important optimizations: Heap properties (e.g. existence of access permissions to a certain heap location) can often be established without calling the theorem prover, because the necessary information is available directly from the symbolic state. Also, heap properties are not included in the assumptions sent to the prover, as they are often costly (e.g. quantifications are needed to express heap updates). In general, these optimizations shift some work from the prover to the verifier and thus improve efficiency. However, they introduce incompletenesses, since path conditions can contain information about the heap (e.g. object aliasing), which is then not reflected by the verifier's decisions, or the heap may contain information that is indispensable for the prover (e.g. object distinctness can be derived from permissions). Syxc solves these problems using several techniques to transfer information between heap and path conditions.

Goals

The goal of this project is to develop a debugger for Syxc that shows possibilities of how to help programmers in finding the reason why a verification fails. This includes finding a suitable (graphical) representation of symbolic state, identifying common debugging situations, identifying helpful interactions between developer, debugger and verifier, and implementing a corresponding GUI prototype.

Scope

The project is divided into a core and extensions. The core is a mandatory part of the project, whereas (a subset of) the extensions are implemented depending on the course of the project.

Core

1. A concept for the resulting debugger tool is to be developed in the form of user stories describing the potential features, discussing their use cases and how users can benefit from them and showing (using UI mockups) how the corresponding GUI elements could look and how the interaction with the user could work.
   The concept contains at least the documentation of one feature of each of the following categories:
   - Heap visualization
   - State manipulation
   - Stepping
   - Answering the questions "Why do/don't I have permissions here"
2. A tool is implemented on top of Syxc supporting a subset of the proposed features. The selection of the features to be implemented is made by the supervisors and depends on the time they require to be implemented and how promising each of them is in terms of the benefit for the users to successfully verify their software.
   The tool supports the same subset of Chalice as Syxc. It is implemented in Scala [5]. It is designed with extensibility in mind and well-documented.

Extensions

- Adding more features to the concept, e.g. of the following categories: State diffs, state querying
- Implementation of more of the proposed features
- Integration of the GUI into an IDE (e.g. Eclipse or VisualStudio)

Deliverables

1. Master's Thesis Report
   Includes the concept for all features proposed in the first part of the core (not just the implemented ones)
2. Project source including documented code, test cases and information about dependencies and how to build and run the project

References

[1] J. Smans, B. Jacobs and F. Piessens, "Symbolic Execution for Implicit Dynamic Frames," 2009.

[2] K. R. M. Leino and P. Müller, "A Basis for Verifying Multi-threaded Programs," *ESOP,* vol. 5502. Lecture Notes in Computer Science, pp. 378-393, 2009.

[3] J. Smans, B. Jacobs and F. Piessens, "Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic," *ECOOP,* vol. 5653. Lecture Notes in Computer Science, pp. 148-172, 2009.

[4] M. Schwerhoff, "Symbolic Execution for Chalice," 2011.

[5] M. Odersky, "The Scala Language Specification," 2011.