Verification of Finite Blocking in Chalice

February 23, 2015

Robert Meier
Supervisor: Prof. Peter Müller

Background

To verify the correctness of programs that use blocking operations, such as acquiring locks, or receiving messages on channels, one often relies on all threads being able to make progress. However, in a setting with non-terminating processes these analyses may unfortunately break down, since threads could potentially postpone the execution of **un**blocking operations forever. To tackle this problem, the concept of *obligations* has been developed [1].

The main idea is that each blocking operation comes with an obligation to perform the corresponding unblocking operation eventually. A measure, that is associated with each obligation, enforces that the satisfaction happens within finitely many steps and is not postponed indefinitely.

In this project we will look at existing extensions to the *Viper* framework [2] that allow the modular verification of finite blocking. In particular we will redesign and simplify the extensions made to *Chalice*, *Silver* and *Silicon*.

Core Requirements

Rethink the Design of Obligations

In light of recent changes in the original design of obligations, it is now appropriate to rethink the original design of the existing Chalice, Silver and Silicon extensions. The new design should consider the following requirements:

- Reduce the necessary Silver extensions to encode the concept of obligations
- Increase the modularity in Chalice
- Allow states where both obligations and credits for a resource exist at the same time

After the redesign all changes will be implemented in Chalice, Silver and Silicon.

Deadlock Detection

Design and implement an encoding of Chalice operations that change the wait-level of a thread, such as fork, join, share, channel creation or blocking operations in general. A global wait order will be introduced to verify that there exist no cyclic wait conditions among threads, which will result in deadlock - free programs.

Extensions

Several extensions are possible. This mainly depends on the progress on the core requirements.

- Improve the error reporting in Silicon
- Devise an extension to make it possible to join a thread more than once
  - In particular, investigate what happens to permissions that are returned by terminating threads
- Design and implement the verification of other operations / liveness properties (for example: read-write locks)
- Make it possible to use more general well-founded sets to describe the lifetime expressions of obligations.
  - Use heuristics to automatically infer the measures associated with an obligation

Schedule

Task	Description	Time
Core	Acquiring relevant background knowledge, reading related work, understanding the problem in detail, familiarization with technical environment	2 weeks
Core	Report: Introduction, Background, Related Work	1 week
Core	Design and implementation of silver extensions + removal of unnecessary extensions	7 weeks
Core	Design and implementation of deadlock checking	3 weeks
Core	Evaluation of implementation + Report	1 week
Extension	Design and implementation of extensions	5 weeks
Extension	Evaluation of implemented extensions + Report	1 week
-	Buffer week	1 week
Core	Finalization of report	2 weeks
Core	Preparation of final presentation	1 week

References

[1] P. Boström and P. Müller, "Modular verification of finite blocking in non-terminating programs," Tech. Rep., ETH Zurich, 2014.

[2] U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," Tech. Rep., ETH Zurich, 2014.