# Verification Condition Generation for the Intermediate Verification Language SIL

## Problem Description

Stefan Heule
stheule@student.ethz.ch

February 22, 2013

## 1 Background

The Semper project attempts to build an automatic verifier for the Scala programming language. In order to achieve this goal, the Semper Intermediate Language (SIL for short) has been designed, which aims to serve as an intermediate verification language that a variety of tools can interact with. For instance, a verifier for a high-level programming language might choose to translate its input to SIL and find appropriate encodings for language features that are not available in SIL. Then, the SIL program can be passed on to a SIL verifier such as Silicon, which will attempt to verify the SIL program, or produce a list of possible errors. These errors can then be translated back to the original input language.

The project chalice2sil implements exactly this approach for the input language Chalice (a small research language for concurrency). Furthermore, a project is currently in progress to translate (a subset of) Scala to SIL.

SIL is a simple object-based language with basic constructs such as methods, loops, and conditionals. Furthermore, it allows one to write specifications using pre- and postconditions, loop invariants, assertions and assumptions. Furthermore, SIL provides a permission model and allows specifications to be written in the style of implicit dynamic frames. To allow transfer of permissions, SIL provides `exhale` and `inhale` statements, and makes it possible to abstract over permissions and values using abstract predicates and abstraction functions.

## 2 Main task

The core goal of this project is to design and implement a verification-condition-generation-based verifier for SIL. This verifier will serve the same purpose as Silicon, namely to verify SIL programs. However, while Silicon is based on symbolic execution, we take the different approach of verification condition generation. For this, we define a translation from SIL to Boogie, which will generate a verification condition that is in turn passed to an SMT solver such as Z3.

We have already gained experience in implementing verification-condition-generation-based verifiers when working on Chalice, a verifier for the programming language of the same name. In this project we want to build on this knowledge and make improvements over that verifier in several key respects. In particular, the new verifier should be built to allow for easy experimentation with alternative Boogie encodings of particular aspects. Ideally, most parts of the verifier should be pluggable in the sense that we can replace them quickly or define a different Boogie encoding for it.

Furthermore, we know certain things that don't work very well in a Boogie based verifier, that we systematically want to avoid. For instance, a problem that we have encountered numerous times in Chalice is the use of quantifiers without explicit triggers. This should be avoided wherever possible in the new verifier.

SIL is conceptually at a different level of abstraction compared to Chalice and will potentially pose different trade-offs and design decisions. Furthermore, the SIL language also has not been fully finalized and the semantics of certain features have to be fully defined first as part of this project.

The core goals for this project are:

- We have a working verifier that takes SIL as input and outputs a list of verification errors, or a message that the program was fully verified.
- The verifier is configurable at run-time in how it translates various constructs. The configuration can occur explicitly by the user through command-line parameters, or based on the input program.

  For instance, if a SIL program does not use epsilon permissions, then a simpler encoding for permissions could be used instead of having the full-fledged pair encoding of permissions.
- The verifier comes with a test-suite that consists of SIL programs with annotations about expected verification outcomes.
- The `goto` statement of SIL allows a program to jump to another location (only forward jumps and jumps out of the loops are allowed, though). This construct is useful to encode `break` or `continue` statements, or exceptional control flow.

  However, it has not been decided what proof obligations need to be met when a `goto` statement is used to exit a loop. In this project, we want to define the semantics of the `goto` statement in the SIL language and implement a verification strategy for it in our verifier.
- The SIL language has been extended by mathematical sequences and a verification strategy is implemented in our verifier.

# 3 Further work

Depending on the time left, progress of the project and personal preferences, there are several extensions possible to the project, including the following:

- **Support for important abstract data types such as sets and multisets.** These data types are useful to model many things and SIL as well as the tools building on SIL should have good support for them. For this reason, these features should be built-in such that tools can support them natively if they want. For the VCG verifier we will most likely rely on an axiomatization, because Boogie does not have any native support for sets or multisets.
- **Support for user-defined abstract data types.** It would be useful to extend SIL with support for user-defined data types, similar to what Dafny allows. This extension would involves extending the SIL language with abstract data types, operations on them (such as pattern matching) as well as a translation to Boogie in order to support verification of abstract data types.
- **Permissions under quantification 1.** At the moment, accessibility predicates are not allowed to occur under quantifiers, but for various examples it would be useful to do exactly that. This extension allows accessibility predicates to occur under quantification over sets. Since sets do not contain duplicates, adding support for this type of quantification is simpler than supporting permissions under quantification in general.
- **Permissions under quantification 2.** This extension explores exactly what kind of patterns of quantifications over permissions besides quantification over sets are useful and how these can be supported in a VCG verifier.
- **Different specification style using permissions under quantification.** At the moment we use abstract predicates to abstract over many locations, e.g. to specify a linked list. This works well when the traversal of the data structure happens in the same manner as the structure of the predicate. If this is not the case, one usually has to write ghost code to guide the prover and the approach becomes less natural to use.

  It would be interesting to explore if we can write specifications in a different way using sets and quantification over permissions to avoid these problems. Because sets are unstructured, it might be possible to prevent the problems of abstract predicates described above.
- **Richer permission expressions.** The current permission system allows to express various operations on permissions included addition and subtraction. However, it is not possible to sum over an unbounded number of permissions. It would be interesting to support such sums and think about other features that are missing from the current permission expressions.

- **Automatic support for (more) functions.** Abstraction functions are useful for specifying programs, and in many cases they are recursive. However, recursive definitions are inherently difficult to support in an automatic verifier, because the prover might unfold the recursive definition infinity often. For abstraction function we have a solution if the function traverses a data structure by unfolding one or more predicates. In this setting, the `fold` and `unfold` statements can be used to guide the prover in when to expand definitions.

  However, in the equally interesting case where this is not the case, there is no solution available. An example for such a function is a recursively implemented factorial function.

- **Predicates with arguments.** Abstract predicates in separation logic typically have one or more parameters. In our setting of implicit dynamic frames, some of these parameters are replaced by abstraction functions, and we know how to translate some of the other parameters to ghost fields. However, it would be interesting to support parameters to predicates in SIL.

It is not necessary to decide at this point which of these extensions should be considered, as this can be best determined after at least part of the main task has been completed.