Verification Condition Generation for the Intermediate Verification Language SIL

Master Thesis Report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://www.pm.inf.ethz.ch/

By: Stefan Heule
   stheule@student.ethz.ch

Supervised by: Dr. Alexander J. Summers
              Prof. Dr. Peter Müller

Date: July 31, 2013
## Contents

1 Introduction 4

2 Background 5
   2.1 Boogie ................................................................. 5
       2.1.1 Assumptions and Assertions: Encoding Verification Conditions ... 5
       2.1.2 Maps .............................................................. 5
   2.2 Semper Project .................................................... 5
       2.2.1 Overview of Semper .............................................. 6
       2.2.2 SIL Front-ends .................................................. 6
       2.2.3 SIL Back-ends .................................................. 7
       2.2.4 Other Tools Interaction with SIL ............................. 7
   2.3 SIL ................................................................. 7

3 The Architecture of Carbon 9
   3.1 Design Goals ....................................................... 9
   3.2 Module System ..................................................... 9
   3.3 Indirect Wiring of Modules: Components ......................... 10
       3.3.1 Ordering of Components ........................................ 10
       3.3.2 Implementation of Components ................................ 11
       3.3.3 Wrapping a Translation ........................................ 11
   3.4 Evaluation of Extensibility and Flexibility ...................... 11
       3.4.1 Alternative Encodings ......................................... 12
       3.4.2 Changes to SIL ................................................. 12
       3.4.3 Cooperating Modules ......................................... 12

4 Translation from SIL to Boogie 14
   4.1 General Remarks on the Translation .............................. 14
   4.2 Main Module ....................................................... 14
   4.3 Statement Module .................................................. 15
   4.4 Expression Module ................................................ 16
   4.5 Heap Module ...................................................... 16
       4.5.1 Encoding of Newly Created Objects ......................... 16
   4.6 Inhale and Exhale Modules ....................................... 17
   4.7 Type Module ........................................................ 18
   4.8 Sequence and Set Module ......................................... 18
   4.9 State Module ...................................................... 18
   4.10 Domain Module .................................................... 18
   4.11 Permission Module ................................................ 18
       4.11.1 Encoding of Permissions ..................................... 18
       4.11.2 Abstract Read Permissions ................................... 19
   4.12 Function and Predicate Module ................................. 20
       4.12.1 Functions ..................................................... 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.12.2</td>
<td>Predicates</td>
<td>21</td>
</tr>
<tr>
<td>4.12.3</td>
<td>Predicates with and without Arguments</td>
<td>23</td>
</tr>
<tr>
<td>4.12.4</td>
<td>Additional Unfoldings</td>
<td>23</td>
</tr>
<tr>
<td>4.13</td>
<td>Readability of the Boogie Output</td>
<td>24</td>
</tr>
<tr>
<td>4.13.1</td>
<td>Identifiers</td>
<td>24</td>
</tr>
<tr>
<td>4.13.2</td>
<td>Comments in the Output</td>
<td>25</td>
</tr>
<tr>
<td>4.14</td>
<td>Optimizing the Output</td>
<td>25</td>
</tr>
<tr>
<td>4.15</td>
<td>Example translation</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparison with Chalice</td>
<td>28</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparison with Silicon</td>
<td>28</td>
</tr>
<tr>
<td>6</td>
<td>Additional Work</td>
<td>31</td>
</tr>
<tr>
<td>6.1</td>
<td>SIL Abstract Syntax Tree Implementation</td>
<td>31</td>
</tr>
<tr>
<td>6.2</td>
<td>SIL Language Definition and Pretty-Printer</td>
<td>31</td>
</tr>
<tr>
<td>6.3</td>
<td>SIL Parser and Type-Checker</td>
<td>31</td>
</tr>
<tr>
<td>6.4</td>
<td>SIL Syntax-Highlighting</td>
<td>31</td>
</tr>
<tr>
<td>6.5</td>
<td>SIL Extension: Unique Constants</td>
<td>31</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>33</td>
</tr>
<tr>
<td>7.1</td>
<td>Status of the Implementation</td>
<td>33</td>
</tr>
<tr>
<td>7.2</td>
<td>Future Work</td>
<td>33</td>
</tr>
<tr>
<td>7.3</td>
<td>Conclusions</td>
<td>33</td>
</tr>
<tr>
<td>7.4</td>
<td>Acknowledgements</td>
<td>34</td>
</tr>
<tr>
<td>A</td>
<td>SIL Language Definition</td>
<td>38</td>
</tr>
</tbody>
</table>
1 Introduction

Software correctness is an important aspect when building systems and is particularly important when software controls critical systems. Testing is a widely used technique to ensure software quality, but it cannot offer any guarantees. Static software verification on the other hand proves that programs behave in the specified way under all circumstances.

Full functional static verification is a challenging task. For this reason, a common approach is to break the task up into smaller problems by using an intermediate verification language. The program to be verified is then translated into the intermediate language, which is then proved correct using a verifier for that language. This approach has many advantages, such as reusing the same intermediate language to verify different host languages. Furthermore, the fact that the verification task is broken up into smaller sub-problems allows one to build simpler tools that perform a clearly defined task well.

The Semper project\(^1\) at ETH Zurich has the goal of building a static verifier for the programming language Scala. As part of this project, the Semper Intermediate Language (henceforth SIL) has been designed as the intermediate verification language. In this Master thesis we designed and implemented a static verifier for SIL, with a particular focus on extensibility. The resulting verifier is called Carbon.

Sec. 2 gives background information on related projects and introduces some of the underlying techniques in software verification. Sec. 3 describes the software architecture of Carbon and Sec. 4 presents how the verification of SIL works in detail. Sec. 6 mentions some additional work of this Master’s thesis and we conclude in Sec. 7.

\(^1\)See [http://www.pm.inf.ethz.ch/research/semper](http://www.pm.inf.ethz.ch/research/semper).
2 Background

In this section we describe some related projects as well as projects that Carbon builds upon. The background on Boogie in Sec. 2.1 is largely taken from [Heu11].

2.1 Boogie

Boogie [ByECD+06] is an intermediate verification language and is used by various static verifiers. It also abstracts over the interfaces of several theorem provers. It is an imperative language that allows the easy encoding of verification conditions.

In this section we describe a small subset of the features of Boogie. The full documentation can be found in [ByECD+06, Lei08], but the description here will suffice to follow the rest of this report.

2.1.1 Assumptions and Assertions: Encoding Verification Conditions

Assertions are program statements used to prescribe properties that need to be proven. For instance, a field access o.f might result in an assertion that the receiver is not null, i.e.,

\[
\text{assert } o \neq \text{null};
\]

On the other hand, an assume statement is used to introduce an assumption, that is information that the prover can use. It has the effect of rendering traces where the assumption would not hold infeasible. For instance, assumptions can be used to constrain the range of values of a variable at a given program point. Boogie allows one to assign an arbitrary value to a variable using the havoc statement. An assumption can then be used to limit the range of possible values that this variable can take. For example, we can assign an arbitrary value between 0 and 200 to the variable i using the statements

\[
\text{havoc } i; \text{ assume } 0 \leq i \land i \leq 200;
\]

2.1.2 Maps

Boogie supports polymorphic map types to denote updateable maps. Such a map type consists of a number of domain types, and a single range type. Furthermore it is possible to have one or more polymorphic type arguments. The domain types are written comma-separated inside square brackets followed by the range type. For instance [int]bool denotes a map from integers to boolean values. For instance, a typical use of such maps is to encode the heap as a map from object references and field names to values.

Carbon uses maps for several things, including to encode the program heap. A simple encoding of a heap is shown in Listing 1. We first define a type ref for references and a polymorphic

```boogie
type ref;

type Field a;

type HeapType = <a>[ref,Field a]a;

var Heap: HeapType;
```

Listing 1: An encoding of a program heap as a map in Boogie.

type Field a for fields of type a. Then, our heap is a map from references and fields to a value of the correct type. These maps can be read and modified, as is shown in Listing 2.

2.2 Semper Project

The overarching goal of the Semper project is to build a static verifier for the Scala programming language. To make such a big project practical, and in order to build a modular verifier,
2 Background

```java
const unique field_f: Field (int);
var obj: ref;
/* .. */

// read value of obj.f
var i: int;
i := Heap[obj, field_f];

// set value of obj.f to 2
Heap[obj, field_f] := 2;
```

Listing 2: Updating and reading maps.

the verification process is split into several distinct phases. At the center of the Semper project stands the Semper Intermediate Language, an intermediate verification language. The verification roughly proceeds by translating Scala code to SIL and then verifying that SIL program.

2.2.1 Overview of Semper

The verification of programs in the Semper project works by translating the input program (which, in theory, can be in any language) to SIL using a front-end, verifying that program using one of the back-ends and potentially transforming the SIL program in-between in some way.

Fig. 1 shows an overview of the existing tools in the Semper project.

![Figure 1: Overview of the Semper project.](image-url)

2.2.2 SIL Front-ends

The task of a SIL front-end is to take input in a particular programming language and produce an appropriate SIL program that, when successfully verified, implies the desired properties of the original program: usually some form of correctness with regards to a given specification. The result of a SIL front-end is always a SIL abstract syntax tree (AST for short) representing a valid SIL program.

Three front-ends have already been developed for SIL. One is a simple parser and type-checker for the SIL language itself that just produces the AST of a SIL program given as a string. This
front-end is mainly used for testing and experimentation. Secondly, there is \textit{scala2sil} [Bro13], a front-end that translates a subset of Scala programs that are annotated with contracts to SIL. Finally, \textit{chalice2sil} [Kla13] is a front-end for a subset of the Chalice language [LM09, LMS09, LMS10], an experimental class-based research language for verifying concurrent programs.

2.2.3 SIL Back-ends

A SIL back-end is a verifier for SIL programs; a back-end takes a SIL program (as an AST) and tries to verify the proof obligations in the given program. The result is either a successful verification or a list of potential errors where the verification failed.

At the moment there are two back-ends. The first one is Silicon [Sch10], which uses a symbolic execution based approach.

The second back-end is Carbon, that has been developed as part of this Master thesis. Unlike Silicon it is based on VCG, where the program to be verified is converted into a logical formula that is then given to a SMT solver such as Z3 [dMB08].

2.2.4 Other Tools Interaction with SIL

Because the verification of a program in the Semper project always goes through SIL, it is possible to build tools that work on the SIL AST and reuse them in a variety of context without having to modify that tool. For instance, such a tool can be used in the verification of Chalice programs using chalice2sil, but also when verifying Scala code with scala2sil.

One example of such a tool would a slicer that takes a SIL program and removes part of that program which is not relevant for a given task. Another example is the static analyser SAMPLE [FM12] that can infer partial specifications for a given SIL program using abstract interpretation.

2.3 SIL

SIL is an object-based language targeted at static verification. It uses assertions such as pre- and postconditions as well as loop invariants to specify the expected program behaviour. If desired, these assertions can be checked by a SIL verifier, which we call a back-end in the Semper project.

In this section we briefly introduce some of the methodology behind SIL and the different language constructs that SIL uses. The full syntax of SIL as a BNF grammar is given in Appendix A.

Objects and the heap. SIL does not have classes like other object-based languages, but rather uses a single type \textbf{Ref} for expressions that evaluate to objects on the heap. It is possible to declare fields in a SIL program, and all of these fields are available for all objects. It is the job of a front-end to ensure that only the actually existing fields are accessed on a given object, if the source language has such restrictions.

Framing. SIL’s methodology is based on implicit dynamic frames [SJP09], which is an approach for modular specification and verification of object-based programs. An important problem in static verification is the frame problem, which is to decide what parts of the state a method may modify and which parts are guaranteed to be left unchanged. In implicit dynamic frames, this problem is solved by using accessibility predicates in specifications to indicate the parts a method depends on and modifies. In SIL, we write \texttt{acc(o.f, p)} to indicate that a method requires the permission \texttt{p} to the field \texttt{f} of object \texttt{o}.

Permissions. The permission model of SIL is based on that of Chalice [HLMS13]. In this model, permissions are fractional [Boy03] where 0 means no permission, 1 represents the full
permission which allows reading and writing a location, and any value in between constitutes a read permission granting read access to a given location.

**Data abstraction.** SIL uses abstract predicates [PB05] to abstract over heap locations. A predicate is a parametrized assertion that can include accessibility predicates, and its definition can be recursive, allowing one to abstract over an unbounded number of locations. In SIL, one holds permission to predicate instances (i.e., the name of the predicate with all its arguments instantiated), and it is distinguished between holding the predicate instance, or the body of that instance. The transition between the two states happens explicitly using the two ghost statements **fold** and **unfold**.

**SIL domains.** SIL provides user-defined domains, which are a collection of uninterpreted functions as well as axioms describing properties of these functions. A domain also introduces a possibly generic type, which allows one to encode custom types.

**Inhaling and exhaling.** SIL uses **inhaling** and **exhaling** to model permission transfer. Intuitively, inhaling an expression means to assume all pure assertions from that expression, and to gain all permissions mentioned. Exhaling is the opposite operation, which asserts all pure assertions and, for all accessibility predicates in the expressions checks that enough permission is available and then gives the permission up. These two operations are statements in the language, but can also be used to encode permission transfer at method calls, where the callee’s precondition is exhaled (and thus permission transferred to the callee), followed by inhaling the method’s postcondition.
3 The Architecture of Carbon

In this section we describe the design goals and some of the implementation choices of the software architecture of Carbon.

3.1 Design Goals

SIL and Carbon should not be seen as a static or fixed language and verifier, respectively. Rather, both are research projects that might change in various ways or get extended in the future. An important aspect when designing language or verifier extensions, or changing existing aspects, is experimentation to compare different approaches and designs. Carbon should therefore be built in a way that makes it easy to do exactly that. There are two major ways in which one might want to experiment using SIL and Carbon.

1. Extend SIL with a new feature that needs to be supported by Carbon.
2. Experiment with different encodings from SIL to Boogie for a given feature of SIL (either a new one, or an existing one).

Both of these should be supported well by the architecture of Carbon, making it easily extensible and flexible.

3.2 Module System

At the core of Carbon’s architecture is a module system. The different aspects of the verification engine are split into different parts that we call modules. Those modules have a clearly defined purpose and each takes care of one aspect of the verification. For instance, one module defines the heap representation and translates heap lookups, while another module performs the exhale operation. Different modules still need to interact and work together as one verifier. This happens through well-specified interfaces, such that multiple different implementations for a single module can exist.

At the moment, Carbon consists of the following module interfaces with one or several implementations for each of them:

- The main module is the starting point of the verification, and additionally translates method declarations.
- The statement module translates SIL statements.
- The expression module handles the translation of SIL expressions.
- The heap module is concerned with the encoding of the heap.
- The inhale and exhale module perform inhale and exhale operations.
- The type module translates SIL types to Boogie.
- The sequence module handles mathematical sequences.
- The set module deals with mathematical sets.
- The state module handles all contributions to the program state (such as the heap, or the current permissions) and implements functionality to support old expressions.
- The domain module translates SIL domains.
- The permission module is concerned with the permission model of SIL.
• The function and predicate module handles functions and predicates.

A detailed explanation of how the individual modules work is given in Sec. 4.

3.3 Indirect Wiring of Modules: Components

Many of the modules need to interact with one another. For instance, the exhale module might ask the heap module to translate a field access in order to assert a certain property about that field. A simple approach to implement this interaction is to explicitly wire the modules together. In the previous example, the exhale module could have a reference to the heap module and when translating a field access call the appropriate method of the heap module.

While this approach works, it is very explicit and does not scale well. Furthermore, it creates problems when the verifier needs to be extended, as changes to existing modules become more involved. In many cases there is one module that performs a certain task and implements the basic functionality, but several other modules need to participate in the translation to handle the features of SIL that they are concerned with. The example from before with the exhale module illustrates this well; the exhale module implements some basic functionality like exhaling boolean connectives, but exhaling the more specialized expressions (such as accessibility predicates) need to be handled by other modules such as the permission module. To accommodate this use-case as well as getting rid of the explicit wiring, we introduce the notion of components and component registries. Any module can serve as a component registry by defining a component interface and providing a way for other modules to register themselves; other modules can then implement the interface and register themselves before the verification starts. In the running example, the exhale module defines an exhale component interface and serves as a registry for components of that type. The exhale module takes care of the basic infrastructure for the exhale operations such as havocing the heap in an appropriate way, deciding which phases are necessary for a given expression, or exhaling logical connectives such as conjunction or implication. How other SIL expressions are exhaled is left to the appropriate modules by just iterating through all registered exhale components and asking them to exhale the given expression. Typically only one component will know how to exhale any given expression, but this system does allow multiple components to contribute to the translation. Note that the exhale module does not know, or need to know, which exact modules registered themselves as an exhale component, which makes the wiring implicit.

This component system is used throughout Carbon, and in particular the following component interfaces exist:

• The inhale and exhale components allow modules to define how a given expression is inhaled or exhaled.
• The statement component allows modules to contribute to the translation of statements.
• The state component allows modules to contribute to the program state.
• The definedness component allows modules to add checks for well-formedness of expressions (e.g., no division by zero).

3.3.1 Ordering of Components

Sometimes, if multiple components contribute to the translation of a certain SIL construct, it can be important that the different contributions are ordered in a certain way. To accommodate for this, modules can indicate partial ordering constraints when registering themselves as a component. For instance, a module might indicate that it would like to get handled before or after a certain other module, or that it should be the first or last component that is considered.
For instance, the statement module asks all components to translate a new statement such as 
\( o := \text{new}() \), and several modules will contribute to the translation. In particular the heap module prepares a new memory cell in the heap using the following Boogie code (details about the \texttt{#alloc} field are explained in Sec. 4.5):

\[
\begin{align*}
\text{havoc freshObj;} \\
\text{assume (freshObj} \neq \text{null)}} \land \neg \text{Heap[freshObj, #alloc];} \\
\text{Heap[freshObj, #alloc] := true;} \\
\text{o := freshObj;}
\end{align*}
\]

Furthermore, the permission module adds permission to all fields, which might produce the following code:

\[
\begin{align*}
\text{Mask[o, f1] := FullPerm;} \\
\text{Mask[o, f2] := FullPerm;}
\end{align*}
\]

In this example it matters which of the two partial translation appears first in the final output; the verification only works correctly if the heap module is handled first such that it can prepare the new memory cell in the beginning. Otherwise, the permission module will not add permission to the correct heap locations. This is easily achieved using the partial ordering infrastructure introduced above, where the heap module can indicate it would like to be handled as the first component.

\textbf{Conflicting orderings.} It is possible that different modules give conflicting partial orderings, which would be a bug in the implementation. Such errors are detected during initialization time of Carbon (i.e., before any verification starts) and are reported to the user appropriately.

### 3.3.2 Implementation of Components

If a module would like to serve as a component registry, it only needs to define a component interface \texttt{C} and extend the trait \texttt{ComponentRegistry[C]}. This trait will provide a method that other modules can use to register themselves as a component of type \texttt{C}. The trait also includes functionality to take all partial ordering constraints into account and generate a compatible total ordering of all registered components, or produce an error if no such ordering exists.

### 3.3.3 Wrapping a Translation

Sometimes it is useful for a module to generate two pieces of Boogie code that both contribute to the translation of a construct, but have the two pieces wrap any other parts of the translation for that construct (i.e., appear at the beginning and end, respectively, in the final output). For instance, if SIL were to be extended with history constraints for fields, then the module implementing this functionality might want to contribute to the translation of field assignments by first recording the pre-assignment value of the field, then have the normal translation (i.e. the actual field update) happen followed by an assertion of the history constraint relating the old and new field value. This would be possible to implement by registering two statement components and having them appear at the beginning and end, respectively. While this works, it is cumbersome, in particular if the two parts want to share information such as the pre-assignment value of the field in the example above. For this reason, there is built-in support in Carbon for wrapping existing translations.

### 3.4 Evaluation of Extensibility and Flexibility

In this section we try to evaluate the extensibility and flexibility of the architecture of Carbon by considering different scenarios that are likely to happen in the future.
3.4.1 Alternative Encodings

In several cases it is useful to have different encodings for the same aspect of SIL. For instance, one encoding might be more efficient (or more complete, or better in some other aspect) in some cases, while another is more suitable in other cases. It might also be that some encodings are only applicable for a certain class of input programs, but behave better than a more general and always applicable encoding. Also, it is often useful to easily be able to try out several different encodings to evaluate their properties.

In Carbon, alternative encodings can be easily implemented. It is possible to have different implementations for a given module and choose in some way between these implementations. The choice which of module implementation to use can happen in any way, e.g., by considering command-line arguments, a configuration file, or more dynamically depending on the input program (e.g., if the input program does not use a certain SIL feature), the environment or some other factor. Of course, different module implementations can share code if it is convenient. In fact, the full Scala language can be used as long as the appropriate module interface is implemented. For instance, one module implementation might extend another using inheritance and override some of the behaviour, while reusing other behaviour. Common functionality can also be implemented in a trait that is then mixed-in in several implementations.

3.4.2 Changes to SIL

In the future, SIL might be extended with new features which might introduce additional statements or expressions, or modify existing ones. In such a case, Carbon will need to adapt and its architecture should make the necessary changes as easy as possible.

3.4.2.1 Modifying Existing Semantics

If the semantics of some existing SIL feature change dramatically, then Carbon’s architecture can do little to make the changes easy as a reimplementation of the affected parts would be necessary. However, it is more likely that the semantics are tweaked only in some minor way. For instance, there might be an additional well-definedness check for an expression, or an additional proof obligation or assumption about a statement. In these cases, the component system will allow the implementation in an easy way by registering an additional component to take care of the check, proof obligation or assumption. Such a change does not require changing any existing modules.

3.4.2.2 Extending the Language

Let's consider an extension to SIL that adds a new statement (extending SIL with a new expression works analogously). To implement a verification strategy for such an extension, one can create a new module, and implement the statement component interface to handle the translation of the new statement. The module can then register itself with the statement module, making the implementation of the extension possible without changing any existing module.

3.4.3 Cooperating Modules

It is typically necessary to have several modules cooperate to achieve the translation of some part of SIL to Boogie. Sometimes the modules should be coupled more tightly than the respective module interfaces allow, but extending the module interface might not be useful because the additional methods are only sensible to some implementations but not others. In such a case, two or more modules can only be used all at once, because of the interaction that is necessary. Such a scenario is supported by the architecture of Carbon as follows: instead
of directly implementing the module interfaces, the group of cooperating modules can first
define sub-interfaces with the additional functionality they use to interact, and then implement
these interfaces instead. To ensure that these modules are not mixed with other incompatible
implementations, they can perform a dynamic check upon initialization of the verifier using a
type-cast in Scala.
4 Translation from SIL to Boogie

In this section we describe the translation of SIL code to Boogie that is used by Carbon. More precisely, we give more details on the different module implementations of Carbon and how they translate the different SIL constructs.

Unless mentioned otherwise, there is only one module implementation for a given module interface, which we call the default module. In that case, we use module and module implementation interchangeably for conciseness.

4.1 General Remarks on the Translation

The translation is a function from a SIL AST to a Boogie AST. To make the development of Carbon as easy as possible, the Boogie AST written for Carbon uses the same names for language constructs available both in SIL and Boogie; this allows people already familiar with SIL to quickly find their way around the Boogie AST, too. To distinguish the two equally named classes from the two AST’s, Carbon uses the convention to never use SIL classes in an unqualified manner but rather prefix all uses with `sil`, whereas classes from the Boogie AST are always used unqualified. Note that even though the SIL AST is not (directly) in a package named `sil`, Scala’s renaming feature during imports allows this convention nonetheless using

\[
\text{import semper.sil.ast => sil}
\]

Now, `sil.Assign` refers to the SIL assignment statement whereas `Assign` (with the correct import of `semper.carbon.boogie.Assign`) refers to the Boogie assignment statement.

Note that the we describe what steps are necessary to verify a given SIL feature like “inhale the precondition of a given method”. Of course, Carbon does not itself perform the inhale operation but instead produces Boogie code that corresponds to inhaling the precondition. If Carbon really does perform some operation during the translation, we explicitly say so.

We use the function \( [ ] \) to denote the translation from SIL to Boogie (for types, expressions as well as statements) in our descriptions.

4.2 Main Module

The main module is the starting point of the translation. It takes a SIL program and splits the input into a list of fields, methods, domains, and functions. The translation of most of these is handled by the appropriate module (e.g., translating domain definitions is performed by the domain module). However, the main module translates method declarations: every SIL method declaration gets translated into a Boogie procedure where the following steps are performed:

1. The program state is initialized using the state module. That is, the heap is havoced (i.e., assumed to be arbitrary), and the permission mask is initialized to the all-zero mask.

2. The method’s precondition is inhaled.

3. A variable is used to store the current state at this point. This is later used as the “old” state, e.g., when checking the postcondition.

4. Assumptions about method parameters are made, if necessary. This includes information about allocatedness of references (see Sec. 4.5.1).

5. The body is verified by translating all statements with the statement module.

6. The method’s postcondition is exhaled.
4.3 Statement Module

The statement module is concerned with translating SIL statements. It serves as a component registry for the statement components which do most of the actual translation. The statement module takes care of translating some basic statements such as assignments, while loops, goto/label statements and method calls. The translation of assignments is straightforward, resulting in an assignment in Boogie with an additional well-formedness check (see Sec. 4.4 for details on well-formedness checking) for any expressions occurring. Method calls are translated by performing the following steps:

1. Save a copy of the current state in local variables so that one can refer to it later.
2. Check the well-formedness of all actual arguments.
3. Exhale the precondition of the method with the formal parameters replaced with the actual values passed.
4. Havoc all targets of the method call, that is, all local variables (if any) to which the method call’s return values get assigned.
5. Inhale the postcondition using the state from the first step as the old state.

While loops are verified as follows:

1. Exhale the loop invariant.
2. Havoc all local variables that get assigned in the loop.
3. Check the well-formedness of the loop invariant.
4. Non-deterministically choose to check the loop body, or continue without the check. Checking the loop body involves the following steps:
   i) Inhale the loop invariant.
   ii) Check the well-formedness of the loop guard.
   iii) Assume the guard.
   iv) Havoc the local variables defined as part of the loop (SIL only allows declaration of local variables as part of a method or a loop).
   v) Verify the loop body by translation all statements.
   vi) Exhale the loop invariant.
5. Inhale the loop invariant.
6. Assume the negation of the loop guard.

This way of verifying while loops is slightly different than what the Hoare rule for loops might suggest. In Carbon, the verification of the loop body does not happen completely separately from the verification of the surrounding context, as it is stated in the Hoare rule. The reason is that this way, the context in which the loop occurs has not to be fully mentioned in the loop invariant. For instance, local variables that are not written to in the loop body are known to have the same value as at the beginning of the loop, and similar for heap locations (for which a read permission is available).

Goto statements and labels are directly translated to goto statements and labels in Boogie.
4.4 Expression Module

The expression module is the counterpart of the statement module but for SIL expressions instead of statements. Most expressions are handled by other modules, but the expression module translates some basic expressions such as integer and boolean literals, old expressions, conditional expressions, quantifications, equality and inequality as well as boolean and integer operators. Most of these can directly be translated to the corresponding Boogie construct, except for old expressions, where the state module is used to temporarily switch to the old state.

Additionally and unlike the statement module, the expression module is also concerned with checking the well-formedness of expressions by defining a well-formedness component interface and serving as a registry for that component. It also registers itself as such a component and checks for division by zero errors. In the end, the checks from all registered components are aggregated, allowing for several checks for any given expression.

4.5 Heap Module

As the name suggests, the heap module is concerned with the encoding of the heap of SIL. To this end, the module first defines a Boogie type for references called \texttt{ref}, as well as a polymorphic type \texttt{Field} \( \alpha \) for fields of type \( \alpha \). The heap is now encoded as a Boogie map from references and fields of type \( \alpha \) to values of type \( \alpha \). For every field \( f \) of type \( \tau \) in the SIL program, the heap module defines a unique constant \( f\# \) of type \texttt{Field} \( \tau \). With this, a field access \( o.f \) in SIL can then be translated to \texttt{Heap}[\( o \), \( f\# \)]. The heap module also declares a constant of type \texttt{ref} for the null literal and translates the SIL null literal to this constant. Finally, the heap module registers itself as a well-formedness component and adds the check that all values which get dereferenced are non-null.

4.5.1 Encoding of Newly Created Objects

When a new object is created, then the verifier should be able to conclude that the new object is different from all previously existing objects. This is achieved by adding a boolean ghost field \#alloc which indicates whether a given cell in the heap currently points to a valid object, or not. The translation of the \texttt{new} statement in SIL then proceeds as follows; first, a new heap cell is chosen by havocing a reference-typed variable \texttt{freshObject} and assuming that \texttt{Heap}[\texttt{freshObject}, \#alloc] is false (we use a previously unused heap cell) and that \texttt{freshObject} \neq \texttt{null} (the new object is non-null). The full Boogie code for this is as follows:

\begin{verbatim}
havoc freshObj;
assume (freshObj \neq \texttt{null}) \land \neg \texttt{Heap}[freshObj, \#alloc];
\texttt{Heap}[freshObj, \#alloc] := \texttt{true};
o := freshObj;
\end{verbatim}

This ensures that the new object is different from all previously existing objects if we make sure that the prover knows \texttt{Heap}[\( o \), \#alloc] to be true for all such objects. There are four ways to obtain a reference-typed value in SIL, and for all of them we want to ensure the appropriate allocatedness information. In particular, all of these ways either yield null, or an object which has the \#alloc flag set to true. More formally, the following information is assumed in all cases:

\[ \text{validObject}(o) = (o = \texttt{null}) \lor \texttt{Heap}[o, \#alloc] \]

This is achieved as follows:

- Field access for a reference-typed field. For this case, the heap module adds an axiom which states that all such lookups yield allocated objects, or null, i.e.
∀ o, f • validObject(Heap[o, f])

One could imagine using explicit assumptions for every field access that occurs in the SIL program. However, this is difficult because field accesses can occur in expressions, where it is not possible to directly place assumptions. Furthermore, the axiom we use allows the prover to reason about field accesses that don’t occur explicitly in the program.

- Method return values. The heap module contributes to the translation of method calls and adds validObject(r) for all reference-typed return values r.

- Function return values. For every function that returns a reference-typed value, an axiom with the appropriate information is added, stating that an invocation of the function yields a value for which validObject holds.

- Method parameters (when verifying the method body). Similar to method return values, the heap module assumes validObject(p) about all reference-typed parameters p at the beginning of the method verification.

Example. Lets consider the following SIL code to illustrate this behaviour:

```sil
method test(p: Ref) {
  var a, b: Ref
  a := new()
  assert a != p
  b := new()
  assert b != p && a != b
}
```

Both assertions are true, because a newly allocated object is compared to something that either already existed before, or is null. For instance, in the comparison a != p, it is known that a is non-null as it has just been created. Now, either p is null and the assertion succeeds, or it is an object which already existed before the allocation site of a, in which case the assertion holds as well.

Note that this information is not available in the Chalice verifier, which means that this example would not verify in Chalice.

4.6 Inhale and Exhale Modules

The inhale and exhale modules implement the inhaling and exhaling of SIL expressions and leave most of the work to components by serving as a component registry for inhale and exhale components, respectively. However, the translations of boolean conjunction (&&) as well as implication (==>) as well as conditional expressions c?e1:e2 are implemented directly as sequential composition for && and Boogie conditionals in the other two cases. Furthermore, all pure assertions can simply be assume and asserted when they are inhaled and exhaled, respectively.

The exhale module additionally havoc the heap and assumes that all framed values in the new heap keep their value. Exactly which values are framed in some way is decided by the permission module. Note that the exhale module does not havoc the heap if the exhaled expression is pure (i.e., does not mention any permissions). This is useful to keep the number of indirections through heap updates small in code that follows by avoiding any change to the heap in this case. The reason this helps is that when the verifier reasons about a heap lookup Heap[o,f], and the heap variable has changed many times in the Boogie program up to this point, then it has to check whether (o,f) is still the same before and after the update using the framing information provided. At the very least this requires instantiating the framing axiom appropriately. If the heap variable is left unchanged, no such reasoning is necessary.
4.7 Type Module

The type module is responsible for translating SIL types to Boogie types. It translates some basic types such as integers and booleans, and relies on other modules for the translation of all other types.

4.8 Sequence and Set Module

The sequence module translates all sequence-related expressions, while the set module deals with set and multiset expressions. Both implementations are based on an axiomatization from Dafny [Lei10]. In this approach, most sequence, set and multiset expressions are translated to uninterpreted function calls, and these functions are then related correctly to one-another using axioms.

4.9 State Module

The state module gathers all aspects that contribute to the current program state such as the heap or permission mask. To this end, it serves as a component registry for state components. Furthermore, the state module maintains two program states, one current state and one “old” state that typically refers to the pre-state. The state module also provides methods to backup these states and restore them later (e.g. one that has been backed up earlier) or create fresh states. All these methods are implemented by calling the corresponding methods on all components who are responsible for dealing correctly with the request.

4.10 Domain Module

The domain module translates SIL domains, i.e., domain functions, axioms and the possibly generic domain type associated with the domain. Most of the translation is straight-forward; domain functions are translated to uninterpreted Boogie functions, domain axioms get translated to Boogie axioms, the domain type to a (possibly generic) Boogie type, and domain function applications to Boogie function applications. The only exception are unique functions in SIL, which get translated to unique constants in Boogie.

Note that the Boogie axioms have the same triggers as the corresponding SIL domain axiom. If no trigger is specified, then Carbon tries to infer a trigger automatically in the same way as Chalice does for universal quantifications written by the user.

4.11 Permission Module

The permission module implements the permission model of SIL, which is based on [HLMS13]. To this end, the permission module uses a mask that is used to track the currently available permission to any given location or predicate. This mask is also registered with the state module as a state contribution, and is initialized to the zero-mask, i.e., a mask where no permission is available.

4.11.1 Encoding of Permissions

Permissions in SIL are either fractions, epsilons or a combination of both. To encode this in Boogie, permissions are encoded as a pair where the first element stores the fractional part and the second element counts the number of epsilon permissions.\footnote{More precisely, since Boogie does not support pairs directly, a map is used where two constants are used as indices.}
4.11.2 Abstract Read Permissions

When entering or leaving a fresh read permission block, the permission module (in the translator) records which local variables are currently considered abstract read permissions. When an accessibility predicate is exhaled that contains such an abstract read permission, then it is potentially possible to generate assumptions about the value of that abstract read permission. The procedure described in [HLMS13] works by exhaling accessibility predicates in three phases. In the first phase, only fixed permissions are exhaled (where no abstract read permissions occur), while in the second phase all permission expressions are exhaled that contain positive occurrences of abstract read permissions. Finally, in the last phase any remaining permission expression is exhaled. This procedure from [HLMS13] only works for a simplified permission model, which required us to generalize the approach for Carbon. Exhaling an accessibility predicate now works in three steps: first, the permission expression is normalized to simplify the implementation of the steps following. Then, the normalized expression is split into a list of conditional expressions associated with a particular phase, and finally each entry in this list is exhaled.

Permission expression normalization.

The normalization proceeds as follows:

1. Permission subtraction is removed by using addition and integer multiplication. The permission expression $p_1 - p_2$ is replaced by $p_1 + (-1) \cdot p_2$.

2. Permission multiplications are moved to the inside. The expression $p_1 \cdot (p_2 + p_3)$ is replaced by $(p_1 \cdot p_2) + (p_1 + p_3)$.

3. Similarly, integer-permission multiplication is also moved to the inside.

4. Multiple integer-permission multiplications are grouped. The permission expression $n_1 \cdot (n_2 \cdot p)$ is replaced by $(n_1 \cdot n_2) \cdot p$.

This rewriting phase is useful to simplify the splitting of permission expressions that follows. In particular, the resulting expression is guaranteed to be a sum of terms, where each summand can only be a permission multiplication, an integer-permission multiplication or a basic permission (such as a constant, a variable or a function invocation that returns a permission).

Splitting of permission expressions. The normalized permission expression is split into a list of triples $(\text{phaseID}, c, p)$, where the first component indicates the phase in which the permission $p$ should be exhaled, but only if the condition $c$ evaluates to true. The reason the tuple also has a condition, is that it is not easily possible to determine statically to which phase a permission expression belongs in general. For instance, consider the expression $1/2 + n \cdot \text{rd}$, where $\text{rd}$ is an abstract read permission. Depending on whether the integer $n$ is positive or not, the expression belongs to phase 2 or not. For this reason, we include a boolean condition in the tuple and let the theorem prover reason which phase applies.

The splitting of a permission expression $p$ works as follows:

- Fixed permissions (i.e., permissions that do not contain any abstract read permissions) belong to phase 1. That is, the tuple $(1, \text{true}, p)$ is returned.

- Abstract read permissions on their own belong to phase 2, that is $(2, \text{true}, p)$ is returned.

- Integer-multiplication with an abstract read permission of the form $n \cdot \text{rd}$ generate two tuples: $(2, n > 0, p)$ and $(3, n \leq 0, p)$. This reflects that in the first case we deal with a positive occurrence of an abstract read permission, but a negative one in the second case.

- Similarly for permission multiplication $p_1 \cdot \text{rd}$, there are two tuples generated with the condition $c$ that $p_1$ is positive permission and $\neg c$: $(2, c, p)$ and $(3, \neg c, p)$.
• Analogously to the last two cases for expressions of the form \( p_1 \cdot n_1 \cdot \text{rd} \) for some permission \( p_1 \), integer \( n_1 \) and abstract read permission \( \text{rd} \).

• Permission additions of the form \( p_1 + p_2 \) are split recursively.

• Any other permission expression belongs to phase three: \((3, \text{true}, p)\).

**Exhaling each tuple.** In the last step, all tuples are exhaled in the appropriate phase if the condition from the tuple evaluates to true (using a Boogie conditional).

**Example.** To illustrate this approach, let’s consider exhaling the accessibility predicate

\[
\text{acc}(\text{this}.f, \frac{1}{2} \div (\frac{1}{2} \cdot \frac{1}{4})+ n \cdot \text{rd})
\]

First, the permission expression is normalized, which leaves us with

\[
\frac{1}{2} \cdot \frac{1}{2} + (-1 \cdot \frac{1}{2} \cdot \frac{1}{4})+ n \cdot \text{rd}
\]

Next, the permission splitting turns this into four tuples, namely

\[
(1, \text{true},\frac{1}{2} \cdot \frac{1}{2}),(1, \text{true},-\frac{1}{2} \cdot \frac{1}{4}),\frac{n}{2}>0,\frac{n}{2}>0)
\]

\[
\frac{n}{2}>0,\frac{n}{2}>0)
\]

Finally, these tuples are exhaled in the correct phase (given the correct condition holds), which results roughly in the following Boogie code\(^3\):

```boogie
assume rd > 0

// phase 1
perm := 0
perm += 1/2 * 1/2
perm += -1 * 1/2 * 1/4
assert perm \leq \text{Mask}[\text{this}, f]
\text{Mask}[\text{this}, f] -= perm

// phase 2
if (n > 0) {
    assert \text{Mask}[\text{this}, f] > 0
    assume n \cdot \text{rd} < \text{Mask}[\text{this}, f]
    \text{Mask}[\text{this}, f] -= n \cdot \text{rd}
}

// phase 3
if (n \leq 0) {
    assert n \cdot \text{rd} \leq \text{Mask}[\text{this}, f]
    \text{Mask}[\text{this}, f] -= n \cdot \text{rd}
}
```

Note that the approach of splitting permissions during exhale as described here is more general than both what is used in Chalice, and what is described in [HLMS13]. In fact, Chalice would not generate assumptions for permission expressions scaled by an integer.

### 4.12 Function and Predicate Module

The function and predicate module translates abstraction functions and abstract predicates from SIL to Boogie. A major challenge in encoding these features in VCG is that both functions

\(^3\)To simplify the example, we ignore epsilon permissions and use fractions directly as permissions. In the real implementation permissions are tuples as described earlier. Also note that certain trivial conditionals like \text{true} have been removed, which is explained in Sec. 4.14.
and predicates can be recursive and thus one needs to be careful not to provide uncontrolled axioms to the theorem prover. More precisely, theorem provers typically use triggers to control how they instantiate universally quantified expressions, which are a set of expressions that the prover must have syntactically seen (modulo equality) before any instantiations are made. This allows the controlled instantiation of quantified axioms. However, if a definition of a recursive function is translated naively to an axiom, then the prover might be able to instantiate that axiom infinitely often, by using the recursive call in the function body. This is called a matching loop.

To avoid this bad behaviour, we have implemented the strategy introduced in [HKMS13]. This paper also proposes the notation of known-folded permissions to frame locations that have already been folded into a predicate. We have implemented the full strategy of [HKMS13] and refer the interested reader to the paper for full details of the approach. Here, we briefly explain the main ideas.

4.12.1 Functions

A function in SIL gives rise to two functions $f$ and $f'$ on the Boogie level, both of which depend on all arguments of the SIL function as well as on the heap (since they might read heap values in their definition). Intuitively, both have the same meaning, but they are used differently in triggers as we explain shortly. There is additionally an axiom that allows the prover to conclude that the two are equal if and only if $f$ has already been observed.

A definitional axiom is used to relate the uninterpreted function $f$ to its body. If the body mentions recursive calls to the function itself, then these are translated to an invocation of $f'$. The axiom however is triggered by $f$, which allows the prover to unfold function definitions exactly one level down. This is known as limited functions.

A framing axiom is used to indicate that the function invocations of $f$ are still equal even in two different heaps as long as the relevant parts of the heaps are equal. The relevant part is any part that the function reads to determine its result, which is approximated by using the frame of the precondition.

For every postcondition of a function, there is an additional postcondition axiom which states that the postcondition holds for a function call.

4.12.2 Predicates

SIL provides abstract predicates with arguments, which need to be folded and unfolded explicitly through ghost statements. We call a given predicate name together with concrete arguments a predicate instance. Such an instance can be folded, unfolded, inhaled and exhaled.

Carbon needs to track permissions, version numbers as well as known-folded permissions for every predicate instance.

It would be possible to use three maps from predicate instances to versions, permissions and known-folded permissions to track this information. However, Carbon reuses the mask for permissions and stores versions and known-folded permissions in the heap as ghost fields. Since predicates do not have receivers, the information is stored in ghost fields of the null literal.

To this end, Carbon first defines two field identifiers for every predicate instance that are used to store the version and known-folded permissions, respectively. More precisely, for every predicate Carbon defines two functions that map from all arguments of the predicate to a field identifier.

Lets consider the example of a list segment predicate $lseg$ with the following definition:

```plaintext
predicate lseg(start: Ref, end: Ref) {
    acc(start.value) && acc(start.next) &&
```
\begin{verbatim}
(start.next != end ==> 
    start.next != null && acc(lseg(start.next, end)))
\end{verbatim}

The predicate contains all permissions to the \texttt{next} and \texttt{value} fields of a linked list starting at \texttt{start} up to \texttt{end}. Such a predicate gives rise to the following two functions in Boogie:

\begin{itemize}
    \item \texttt{function lseg_version(start: Ref, end: Ref): Field Pred_lseg int}
    \item \texttt{function lseg_predmask(start: Ref, end: Ref): Field Pred_lseg PredicateMaskType}
\end{itemize}

As mentioned, these functions return field identifiers and are used to record information in the heap and the mask. It is clear that these field identifiers are different from the identifiers used for regular SIL fields, and that the functions from different predicates yield different field identifiers. We use the Boogie type system to encode this information as follows: the field type is polymorphic and takes two type arguments. One determines the type of the field (e.g., one that stores an integer) and the other is used to indicate the kind of field. There is one kind for regular fields, and one for every predicate. In the running example, this would look as follows:

\begin{verbatim}
// the type of fields
type Field FieldKind FieldValueType;
// the heap variable
var Heap: [Ref, Field K FieldValueType] FieldValueType;
// field kind for regular fields
type NormalField;
// field kind for the lseg predicate
type Pred_lseg;
// regular field identifier (for the next field)
const unique next: Field NormalField Ref
\end{verbatim}

With these definitions, the prover knows that for all \(a\) and \(b\) that \(\text{lseg_version}(a,b)\neq\text{next}\), and similarly for \(\text{lseg_predmask}\). The only remaining information we need to provide is that if two predicate field identifiers are the same, then the arguments must be the same. That is, in our example we add the following two axiom:

\begin{verbatim}
axiom (\forall \text{start}: \text{Ref}, \text{end}: \text{Ref}, \text{start2}: \text{Ref}, \text{end2}: \text{Ref}
    \text{lseg_version}(<\text{start}, \text{end}) = \text{lseg_version}(<\text{start2}, \text{end2}) \implies
    \text{start} = <\text{start2} \land \text{end} = <\text{end2});
axiom (\forall \text{start}: \text{Ref}, \text{end}: \text{Ref}, \text{start2}: \text{Ref}, \text{end2}: \text{Ref}
    \text{lseg_predmask}(<\text{start}, \text{end}) = \text{lseg_predmask}(<\text{start2}, \text{end2}) \implies
    \text{start} = <\text{start2} \land \text{end} = <\text{end2});
\end{verbatim}

This entails that there is a one-to-one mapping from the arguments of a predicate to their field identifiers.

These definitions suffice to encode predicates with arguments. The version number for the predicate \(\text{lseg}(a,b)\) can now be looked up in the heap as \(\text{Heap}\[null, \text{lseg_version}(a,b)\]\), and the permissions to that predicate instance are stored in the mask as follows: \(\text{Mask}\[null, \text{lseg_version}(a,b)\]\).

\textbf{Ghost operations on predicates.} The operations on predicates are now straightforward to implement. To fold a predicate instance, the body of the predicate is exhaled (with all arguments correctly instantiated), followed by inhaling permission to the predicate instance. Unfolding a predicate is the inverse operation, where permission to the instance is exhaled, followed by inhaling the predicate body. In both cases we additionally generate a trigger that allows to unfold function definitions that depend on this predicate one more level down, as explained in [HKMS13].

\footnote{We arbitrarily chose the version ghost field to store the information. It would equally be possible to use the known-folded permission ghost field.}
**Inhaling and exhaling predicates.** Inhaling a predicate adds permission for the predicate instance to the mask, while exhaling a predicate instance checks that the required permission is available, followed by giving up that permission.

**Non-aliasing information from predicates.** Based on which predicate instances are folded inside one another it is possible to deduce non-aliasing information. For instance, if we know that a predicate instance \( p(a, b) \) is folded inside \( p(c, b) \), then we know that \( a \) and \( c \) cannot be aliases.

Note that this feature has been introduced in Chalice for predicates without arguments (only a receiver). In Carbon we have generalized the idea to predicates with any number of arguments.

### 4.12.3 Predicates with and without Arguments

Chalice only provides abstract predicates without arguments and thus can use a simpler encoding. In particular, there is only a single field identifier per predicate (since there are no arguments, there is no need to use a function) and stores the information as ghost fields of the receiver.

Since even in SIL many predicates only have a single argument (a "receiver"), we wanted to ensure that these simple predicates that were already available in Chalice still perform well in Carbon. To this end, we compared the performance of Carbon with a modified version of Carbon that only supports predicates without arguments and uses an encoding of predicates like Chalice. We ran both versions on all tests from the SIL test suite that don’t make use of predicates with arguments. Every test has been run several times and the running time was averaged to account for variations in the execution time. Overall the performance of the two versions is roughly the same, with the regular Carbon version being 5.04% faster on average. We cannot fully explain why the seemingly more complicated encoded is faster on average, but the difference is not very significant. Fig. 2 shows the detailed results.

### 4.12.4 Additional Unfoldings

One idea that Alexander J. Summers had is to temporarily unfold predicates when predicate instances are inhaled. Effectively this would allow the prover to peek into a predicate one level down, potentially yielding additional information such as non-aliasing information. Consider the following example:

```plaintext
var f: Ref

predicate valid(this: Ref) {
  acc(this.f, write)
}

method test(this: Ref, other: Ref)
requires acc(other.f, write)
  requires acc(this.valid(), write)
{
  assert (this != other)
}
```

The assertion in method `test` is true, since the method holds full permission to `other.f` (directly from the precondition) as well as to `this.f` (as part of the `valid` predicate). Therefore, the two variables cannot be aliases. However, since the `valid` predicate has not yet been unfolded, Carbon will not be able to prove this property. If on inhaling predicates are temporarily unfolded, then this information becomes available to the prover.

We have implemented this such that predicate instances are unfolded in a temporary state (which is thrown away afterwards) upon inhaling that predicate instance. This works as ex-
Figure 2: Comparison of a variant of Carbon that uses a Chalice-like encoding for predicates, and the full version of Carbon that additionally supports predicates with arguments. The y-axis shows the execution time in milliseconds.

expected and the example above now verifies without a problem. However, the performance of Carbon is heavily influenced. On average, the verification time increases by 55.3%, and is even higher for test-cases that make heavy use of predicates. We suspect that the additional temporary state copies provide a performance problem, but it is difficult to exactly pin down what causes the slow-down. For now, this feature is disabled in Carbon.

4.13 Readability of the Boogie Output

Carbon tries hard to make the resulting Boogie code as readable as possible to make it easier to debug and work with Carbon and its output.

4.13.1 Identifiers

One aspect to keep the output easily understandable is that Carbon attempts to keep identifiers such as local variable or procedure names close to their SIL counterpart and if possible free of automatically generated pre- or post-fixes. The modular architecture of Carbon adds an additional challenge to implement this; it should be possible that two modules use the same name for some local variable that they generate in the Boogie output without having them to ever know that there is a different module using the same name. On the other hand, it would be against Carbons goal to keep the Boogie output simple to just prefix the variables by, say, the module name to distinguish them.

The solution to this problem in Carbon is to associate a namespace with every identifier (additionally to the name of the identifier). Two identifiers with the same name but different namespaces are treated as distinct entities and the pretty-printer of the Boogie AST will distinguish them as necessary using sequential numbers appended to the end. This works well
in practice, as identifier names from different modules are typically already distinct, leading to concise and readable Boogie output. Otherwise, the namespace system nicely distinguishes without the programmer having to do anything besides using the correct namespace. Finally, even using the namespaces is hugely simplified by using Scala’s implicit values. Typically, a module uses a single namespace which it declares as an implicit value. When creating identifiers, this value will be used implicitly, without the programmer having to mention the namespace at all.

4.13.2 Comments in the Output

Another important measure employed in Carbon to make the Boogie output more understandable are comments. The Boogie AST provides a series of flexible factory methods for creating comments that are attached to single statements, a group of statements or declarations. The methods take care of correctly handling indentation and newlines to maximize readability and to make clear which statements and comments belong together. Furthermore they make sure to only add the comment if there is actually some Boogie code that follows (and not just the empty statement). For instance, before exhaling the precondition at a method call, there is an appropriate comment in the output. However, if the method does not have any preconditions, then the factory will automatically omit the redundant comment without the programmer having to check if there are preconditions manually. This helps keeping the actual translation code simple and focused on the important aspects (i.e., the actual translation).

4.14 Optimizing the Output

Sometimes it is possible to statically decide certain conditions or partially evaluate expressions in the Boogie AST. This is done using an optimizer that takes a Boogie AST and produces an equivalent AST where as many sub-expressions have been evaluated statically and redundant branches removed. This step might seem minor as a theorem prover would probably be able to decide these steps fairly quickly, but it is actually of great usefulness in Carbon. First of all, it again makes the Boogie output more concise and readable. Furthermore and more importantly, it allows the implementation of Carbon modules to not worry about such low-level optimizations and implement certain things uniformly without attempting to statically evaluate certain trivial things; rather, the module can just uniformly produce a general Boogie encoding and let the optimizer take care of removing redundant parts.

The optimizer currently performs the following transformations:

- Constant folding for arithmetic and boolean operators.
- Removal of dead branches (i.e., branches where the condition is a constant).
- Removal of assertions known to hold.
- Removal of if statements with empty then and else branches.

An example where this optimization is particularly useful is the multi-phase exhaling of permission expressions introduced in Sec. 4.11.2. There, the tripels include a condition, which in many cases can be simplified, or even is true altogether.

Note, however, that also Boogie code that directly comes from the SIL input is optimized. This might be unexpected if a user of Carbon is comparing the Boogie code with the original SIL code. However, we found that this rarely happens, as typically SIL code does not contain trivial conditions unless they are meant for debugging. For instance, one might wrap a piece of code in if (false) to effectively comment it out from a verification perspective. If such code fragments are no longer included in the Boogie output, this might be an advantage.
4.15 Example translation

To illustrate these features, we consider the simple SIL program shown in Listing 3, together with the corresponding output of Carbon in Listing 4. For instance, note that all variable names as well as the method name are preserved in the Boogie output, because there was no need to rename anything. Only if the user were to choose a variable name like `Heap`, then the namespace system would automatically recognise the potential conflict and disambiguate appropriately. Also note that in the generated code, there is nothing to indicate the checking of the postcondition; since there is no postcondition, this part is completely left off.

```plaintext
var next: Ref

method test(this: Ref, i: Int, j: Int)
  requires i > j
  requires acc(this.next, write)
{
  var tmp: Ref
  tmp := new()
  this.next := tmp
}
```

Listing 3: Simple (contrived) SIL program to illustrate the translation to Boogie.
procedure test(this: Ref, i: int, j: int) returns ()
modifies Heap, Mask;
{
    var perm: Perm;
    var freshObj: Ref;
    var tmp: Ref;

    // -- Initializing the state
    Mask := ZeroMask;
    assume state(Heap, Mask);

    // -- Checked inhaling of precondition
    assume i > j;
    perm := FullPerm;
    assume this ≠ null;
    Mask[this, next] := PermAdd(Mask[this, next], perm);
    assume state(Heap, Mask);

    // -- Initializing of old state
    assume Heap = old(Heap);
    assume Mask = old(Mask);

    // -- Assumptions about method arguments
    assume this = null ∨ Heap[this, #allocated];

    // -- Translating statement: tmp := new() -- example.sil:9.5
    havoc freshObj;
    assume freshObj ≠ null ∧ ¬Heap[freshObj, #allocated];
    Heap[freshObj, #allocated] := true;
    tmp := freshObj;
    Mask[tmp, next] := FullPerm;
    assume state(Heap, Mask);

    // -- Translating statement: this.next := tmp -- example.sil:10.5

    // -- Check definedness of this.next
    assert {:msg " Assignment might fail. Receiver of this.next might be null
     . (example.sil:10.5) [7]"}
    this ≠ null;
    Heap[this, next] := tmp;
    assert {:msg " Assignment might fail. There might be insufficient
     permission to access this.next. (example.sil:10.5) [9]"}
    FullPerm[#frac] = Mask[this, next][#frac] ∧ FullPerm[#eps] = Mask[this,
     next][#eps];
    assume state(Heap, Mask);
}

Listing 4: The translation result of Listing 3.
5 Evaluation

To evaluate how well Carbon works compared to other verifiers we ran performance experiments that compare Carbon with Chalice and with Silicon.

5.1 Comparison with Chalice

The Chalice verifier uses Chalice as the input language, and the front-end chalice2sil can translate many Chalice programs to SIL. Unfortunately, at the time of this Master’s theses, no working version of chalice2sil was available, so a comparison using chalice2sil was not possible. However, SIL and Chalice are sufficiently similar as languages that many examples in the Chalice test suite can be directly translated to SIL without much effort. We have written a small tool that automatically translates such examples by performing the following transformations to accommodate for the differences in SIL and Chalice:

- Add explicit **this** parameters to methods, functions and predicates, as these features are static in SIL, but aren’t in Chalice.
- Translate all class-types from Chalice to **Ref** in SIL and rely on the Chalice type-checker that field accesses only refer to fields that exist on the given receiver.
- Use fractions instead of percentages for permissions.
- Small changes in the syntax between the two languages, such as:
  - Predicate instance \( o.p \) in Chalice are translated to \( p(o) \) in SIL.
  - The sequence type \( \text{Seq<T>} \) is written \( \text{Seq[T]} \) in SIL.

Note that many valid Chalice programs are rejected, namely if they use features not directly supported in SIL such as monitors, fork/join or channels. Nevertheless, this simple transformation allowed us to run both Chalice and SIL on a subset of the Chalice test suite and compare their performance.

On average, Carbon is 13.81% faster and the full details of the tests are shown in Fig. 3. One possible explanation is that the start-up time for Carbon is lower than for Chalice, resulting in a small constant advantage for Carbon in terms of execution time. Another possible reason for the slightly better performance of Carbon is that in certain cases Carbon is able to avoid heap modifications where Chalice would change the heap. For instance, exhaling a pure assertion does not require havocing the heap, but Chalice still performs such an operation.

The two test-cases framing-fields and test7 stand out; in the former Carbon is more than twice as fast whereas Chalice takes only slightly more than half the time Carbon requires. It is difficult to say what the reason for this behaviour is without further examples. We manually inspected the two tests, but could not find any characteristic specific to these tests that might explain the difference. Once chalice2sil is ready, more tests will hopefully give a clearer picture of the differences.

5.2 Comparison with Silicon

Silicon is an alternative back-end for SIL based on symbolic execution, which allows for a direct comparison with Carbon on the SIL test suite. We show the detailed comparison in Fig. 4. At the time of this Master’s thesis Silicon only worked on some of the test-cases. Test-cases where Silicon does not perform in the expected way are indicated with a black bar and are not considered in the comparison. Note that the y-axis is cut off at 5 seconds; however, the only two tests that take more time only work in Carbon and are thus not relevant here.
On the tests that work for both verifiers, Silicon typically out-performs Carbon significantly. We are not able to fully explain this behaviour, but there are several possible explanations for at least part of the difference. First of all, Silicon is less complete than Carbon and uses simpler encodings for a variety of SIL features. It seems likely that this allows Silicon to perform better for the tests where the incompleteness is not visible due to the simple encoding. That Silicon is less complete can also be seen in the test suite where several tests fail due to this incompleteness. Examples for simpler encodings in Silicon include the following:

- Silicon does not have support for epsilon permissions, allowing for a much simpler permission encoding. In particular, there is no longer a case-split necessary when checking if any permission is available, which is necessary in case of epsilon permissions.
- Silicon does not use a multi-phase exhale as Carbon does, resulting in incompleteness.
- Silicon has no support for known-folded permissions.

Another problem of the comparison is that many of the SIL tests are very small, unit-test-like examples that are not particularly relevant in practice but useful for testing. We suspect that Silicon can benefit from this where in some cases the prover does not have to be queried at all, whereas Carbon will always generate a Boogie program, pass it to Boogie which will run it through the theorem prover.

Finally, previous work [KMS12] already provides some evidence that symbolic execution might be faster than verification condition generation. In this study, however, the difference was less pronounced that what we are seeing with Carbon and Silicon.
Figure 4: Comparison of Carbon with Silicon on the SIL test suite. The tests where Silicon does not work correctly are not considered and are marked with a black bar. The y-axis shows the execution time in milliseconds.
6  Additional Work

Besides the core of this project and the extension, a number of additional things have been worked on. While these are not directly part of the project, they can still be considered to be part of this Master’s thesis.

6.1  SIL Abstract Syntax Tree Implementation

At the time when we started this work, there already existed an implementation of an abstract syntax tree for SIL in Scala. However, up to that point SIL had already evolved in various ways, which left certain artefacts in the AST implementation. Furthermore, certain design decisions with regard to how the AST should be implemented have been revisited and decided another way recently. Finally, the AST implementation did not yet reflect the decision to use the permission model from [HLMS13].

All of these factors led us to decide that it would be best to reimplement the AST from scratch, which was done in collaboration with Bernhard F. Brodowsky. The new implementation makes heavy use of core Scala features such as case classes, traits and case objects. This led to a decrease in lines of code by roughly an order of magnitude.

In addition to the AST itself, we have also implemented a variety of utility methods to visit an AST, access all subnodes, all free type variables or undeclared local variables, as well as transforming a program, statement or expression by replacing certain nodes.

6.2  SIL Language Definition and Pretty-Printer

At the beginning of the project, only an abstract definition of SIL existed, but no concrete syntax had been chosen. To allow the implementation of a SIL front-end with a parser and type-checker as well as a pretty-printer, we defined a concrete syntax of SIL as shown in Sec. 2.3. Based on this definition we have also implemented a SIL pretty-printer which has simplified debugging a great deal and allows the back-ends to generate better error messages when code snippets (in particular expressions) appear in these messages.

6.3  SIL Parser and Type-Checker

Testing a SIL back-end such as Carbon is ideally done by creating a SIL program with some expected verification outcome and then checking whether the verifier reports the expected result. However, constructing a SIL AST directly is tedious, and for most of the project there were no working SIL front-ends available. To improve this situation, we have implemented a SIL parser and type-checker. This allows us to write test-cases directly in SIL.

6.4  SIL Syntax-Highlighting

To make working with SIL more comfortable, we have defined syntax highlighting for LaTeX as well as for the Scala IDE IntelliJ. An example for the IntelliJ highlighting can be seen in Fig. 5.

6.5  SIL Extension: Unique Constants

In SIL, nullary domain functions can be used to define constants. Sometimes it is necessary to add the knowledge that a group of constants are pair-wise distinct, which can be done with an axiom of size $O(n^2)$ for a group of $n$ constants. To enable a more structured way of supporting this use-case of constants we have extended SIL with unique constants. The semantics of a
unique constant is that it is assumed to be different from all other unique constants of the
same type.
Since some theorem provers support such unique constants, back-ends typically do not have to
handle them manually but let the underlying theorem provers reason about them.

Figure 5: Syntax highlighting for SIL in IntelliJ.
7 Conclusions

7.1 Status of the Implementation

Carbon is implemented in the Scala programming language and supports the full Semper Intermediate Language as input. It can be used as a SIL back-end in any project that produces a SIL AST and wants to verify that AST. Furthermore, it provides a command-line interface to run Carbon directly on an input file containing a SIL program.

The changes to SIL described in this report have also been fully implemented and are available as part of the main SIL repository at https://bitbucket.org/semperproject/sil.

To ensure the quality of the verifier, we have developed a test-suite consisting of SIL programs with annotations that indicate the expected verification outcome. These tests are written verifier-independent and are part of the SIL repository, which allows them to be reused easily by any other back-end. All test-cases pass for Carbon.

7.2 Future Work

There are several directions for future work. For instance it would be very worthwhile to do a more detailed performance comparison with Chalice and Silicon, in particular using more realistic and larger examples. For Chalice this should easily be possible once chalice2sil is working again.

From our preliminary experiments with Carbon and Silicon we also have some ideas how one might be able to improve the performance of Carbon. Namely, it would be interesting to experiment with simpler encodings for a variety of SIL features in cases where a fully general encoding is not necessary. An example for this are epsilon permissions, which are rarely used in our examples. It might be that this speeds up Carbon significantly and is able to make the performance gap smaller.

Another idea that Uri Juhasz mentioned is to try and optimize the Boogie code such that there are less traces through a given Boogie procedure. For instance, the permission splitting during exhaling often produces two conditionals with the condition $c$ and $\neg c$, which could be translated to an if-then-else.

7.3 Conclusions

We have presented and implemented Carbon, a verifier for the Semper Intermediate Language. The verifier is based on verification condition generation and is implemented as a translation from SIL to Boogie. We have applied some known techniques, many of which were inspired by Chalice, but went beyond what Chalice supports and generalized existing techniques to work in more settings. In particular, we have generalized and improved on the two-phase exhaling used in Chalice as well as the three-phase exhaling described in [HLMS13]. Our implementation supports more permission expressions such as permission multiplication or scaled permissions compared to the paper and is more complete than the approach in Chalice. We have also extended SIL with abstract predicates with arguments and implemented a verification strategy for Carbon. Our treatment for abstraction functions and abstract predicates is based on [HKMS13], but extended to work with predicates that have arguments. With this, we provide evidence for the claim in [HKMS13] that the approach also works in a setting where abstract predicates have arguments. We have also extended SIL with mathematical sequences, sets and multisets.

Carbon’s architecture has been built with extensibility and experimentation in mind. The module and component systems allow to easily adapt to changes or extensions in the SIL language, as well as supporting experimentation with different encodings of certain aspects.
Which encoding is used can also be input-dependent, allowing the verifier to choose the most optimal encoding for the given program.

We have compared the performance of Carbon with both Chalice and Silicon, to alternative verifiers for the same or similar languages. In these preliminary experiments, Carbon performs roughly equally well as Chalice, even though it supports more general features such as abstract predicates with arguments or a more general three-phase exhale. Silicon currently out-performs Carbon significantly, and we have several ideas how to improve Carbon to hopefully make the performance gap smaller. It is also not fully clear how relevant the tests from the SIL test suite are for such an experiment, and hopefully a more detailed comparison with larger examples will give a better idea of how Carbon and Silicon compare.

7.4 Acknowledgements

I would like to thanks my supervisor Alexander J. Summers for the many great discussions and his valuable feedback. My thanks also go to Peter Müller for giving me the chance to work on this verification project. Furthermore, I would like to thank Bernhard F. Brodowsky for his collaboration on the SIL AST implementation as well as Malte Schwerhof for discussions and help with Silicon.
List of Figures

1. Overview of the Semper project. ................................................. 6
2. Comparison of a variant of Carbon that uses a Chalice-like encoding for predicates, and the full version of Carbon that additionally supports predicates with arguments. The y-axis shows the execution time in milliseconds. ............... 24
3. Comparison of Carbon with Chalice on a subset of the Chalice test suite. We took all examples that we were able to automatically translate from Chalice to SIL. The y-axis shows the execution time in milliseconds. .................. 29
4. Comparison of Carbon with Silicon on the SIL test suite. The tests where Silicon does not work correctly are not considered and are marked with a black bar. The y-axis shows the execution time in milliseconds. .................. 30
5. Syntax highlighting for SIL in IntelliJ. ........................................ 32

List of Listings

1. An encoding of a program heap as a map in Boogie. ...................... 5
2. Updating and reading maps. ...................................................... 6
3. Simple (contrived) SIL program to illustrate the translation to Boogie. .... 26
4. The translation result of Listing 3. ............................................ 27
5. The syntax of SIL. ................................................................. 38
References


A  SIL Language Definition

```
sil-program ::= ( domain | field | function | predicate | method )^  
  domain ::= "domain" domain-name "{"  
    domain-function'  
      axiom'  
"}"
  domain-name ::= ident | ident "[" ident "]"
  domain-function ::= "unique"? "function" ident "(" formal-arg ")" ":" type
  formal-arg ::= ident ";" : type
  axiom ::= "axiom" ident '{"  
    exp ";"?
  "}"
  field ::= "var" ident ";" : type
  function ::= "function" ident "(" formal-arg ")" ":" type
    precondition'  
    postcondition'
  "{" exp "}"  
  precondition ::= "requires" exp ";"?
  postcondition ::= "ensures" exp ";"?
  invariant ::= "invariant" exp ";"?
  predicate ::= "predicate" ident "(" formal-arg ")" "{" exp "}"
  method ::= "method" ident "(" formal-arg " returns" "(" formal-arg ")")"  
    precondition'  
    postcondition'
  "{" local-decl' stmt "}"  
  local-decl ::= "var" ident ";" : type
  stmt ::= (stmt ";"?)^  
    "assert" exp  
    "inhale" exp  
    "exhale" exp  
    "fold" "acc" "(" loc-access "," exp ")"  
    "unfold" "acc" "(" loc-access "," exp ")"  
    loc-access ":=" exp  // field assignment  
    ident ":=" exp  // local variable assignment  
    "if" "{" exp "}" "{"  
      stmt 
    "}"  
    ("elseif" "{" exp "}" "{"  
      stmt 
    "}")^  // (any number of elseif branches
  )
  ("else" "{" stmt "}")?  // (the else branch is optional)
  "while" "{" exp "}"  
    invariant'
```
A SIL Language Definition

```
"{\"stmt \"})"
| ident ":= \"new()\"" // object creation
| ident "(" exp ")" // method call
| ident ":= ident "(" exp ")" // method call with return
values
| \"goto\" ident // goto statement
| ident ":" // a label (can be used as target
for goto)
| \"fresh\" (ident`) "{" // fresh abstract read permission
block
stmt
"

exp ::= exp "?" exp ":" exp // conditional expression
| exp "==>" exp // implication
| exp ("||" | 
"&&") exp // disjunction and conjunction
| "!" exp // boolean negation
| exp ("==" | 
"\!=") exp // equality comparison
| exp (< | \"<\" | 
">" | 
">=") exp // ordering (both numerical
|//permission)
| exp ("+" | 
"-" | 
") exp // math operators (both numerical
|//permission)
| exp "*" exp // int/int, perm/perm and int/
perm multiplication.
| exp ("\" | 
"\%") exp // integer division and modulo
| ("+" | 
"-" ) exp // math operators (both numerical
|//permission)
| ident "(" exp 
")" // function application
| loc-access // field read or predicate
| integer // integer literal
| \"null\" // null literal
| \"true\" | \"false\" // boolean literal
| ident // local variable read
| \"result\" // result literal in function
postconditions
| \"acc\" (" loc-access 
\", exp ")" // accessibility predicate
| \"forall\" formal-arg 
:." trigger" exp // universal quantification
| \"exists\" formal-arg 
:." exp // existential quantification
| "(" exp 
")" // inhale exhale expression
| perm (" loc-access 
") // current permission of given
location
| \"write\" // full permission literal
| \"none\" // no permission literal
| \"epsilon\" // epsilon permission literal
| \"wildcard\" // wildcard permission
| exp "/" exp // concrete fractional permission
| \"Seq\" \[ type \] "()" // the empty sequence
| \"Seq\" (" exp 
")" // explicit sequence
| (" exp ".." exp 
") // half-open range of numbers
| exp "++" exp // sequence append
| (" exp [" // length of a sequence
| exp "[" exp 
"]" // sequence element for given
index
```
| exp "[ ".." exp "]" // take the some of the first elements |
| exp "[" exp ".." "]" // drop some elements at the end |
| exp "[" exp ".." exp "]" // take and drop at the same time |
| exp "in" exp // element containment test |
| exp "[" exp ":=" exp "]" // sequence with one element updated |

| "Set" "[" type "]" "()" // the empty set |
| "Set" "(" exp ".)" // explicit set |
| "Multiset" "[" type "]" "()" // the empty multiset |
| "Multiset" "(" exp ")" // explicit multiset |
| "|" exp "|" // set/multiset cardinality |
| exp "union" exp // set/multiset union |
| exp "intersection" exp // set/multiset intersection |
| exp "setminus" exp // set/multiset subtraction |
| exp "subset" exp // set/multiset subset test |

domain type |

trigger ::= "{" exp "}" // a trigger for a quantification |

loc-access ::= exp "." ident // field access |
| ident "(" exp ".)" // predicate |

type ::= "Int" | "Bool" | "Perm" | "Ref" // primitive types |
| "Seq" "[" type "]" // sequence type |
| ident // type variable or non-generic |
| domain type |
| ident "[" type "]" // generic domain type |

ident ::= "[a-zA-Z_][a-zA-Z0-9$]*" // an identifier (specified as regular exp)