# Integrating the Abstract Interpreter Sample with the Symbolic-Execution-Based Verifier Silicon

**M.Sc. Thesis Description**

Severin Heiniger
severinheiniger@student.ethz.ch

November 11, 2013

## 1. Background

The Semper project aims to build an automated verifier for concurrent programs written in the modern programming language Scala. Fig. 1 gives an overview of its components.

### 1.1. Verification

The project includes two verifiers: Carbon [7] and Silicon [11], based on verification condition generation and symbolic execution, respectively. These verifiers do not directly operate on Scala programs, but rather on the Semper Intermediate Language (SIL). Hence, Scala [2] (or Chalice [8]) programs need to be translated to SIL before verification.

The project's verification methodology is based on that of Chalice [9], which is itself based on implicit dynamic frames [12] and centers around permissions [1] and permission transfers to handle framing and guarantee the absence of data races.

### 1.2. Specification Inference

Another component of the Semper project is Sample (Static Analyzer of Multiple Programming LanguagEs) [4]. It is a generic static analyzer based on abstraction interpretation. It generically combines abstract heap and value domains. Sample contains a new heap analysis [6] that uses a combined-heap-and-value domain and infers invariants on recursive data structures like linked lists. Sample can also infer access permissions [5].

Sample has its own front-ends that translate Scala and other languages to an intermediate language called Simple.

Sample is currently not integrated with other components of the Semper project. Hence, these components do not benefit from the inference power of Sample.

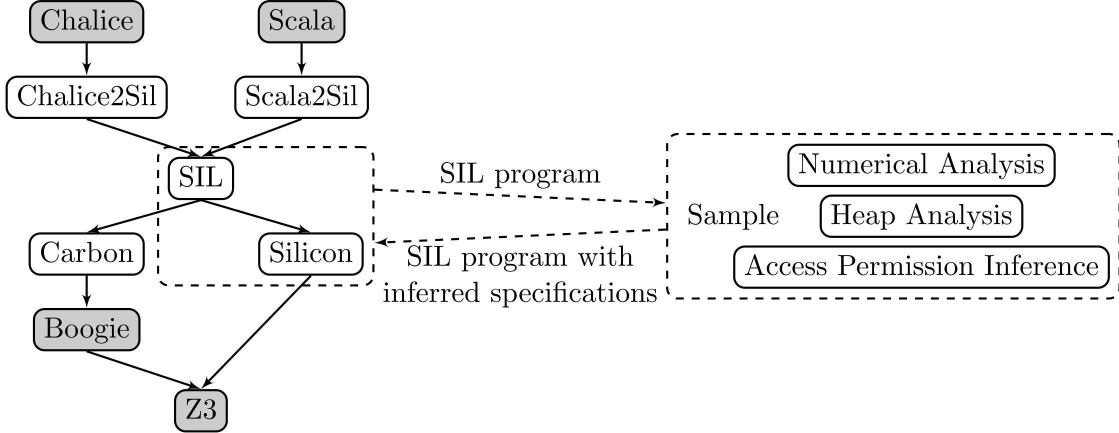


Figure 1: Selected components of the Semper project (white boxes).

## 2. Core Objectives

The goal of this M.Sc. thesis is to integrate the verification and static analysis parts of the Semper project and thus enable a successful verification of incompletely-specified SIL programs. This reduction of annotation overhead is an important step towards the vision of making verification more practical.

Concretely, we want to translate incompletely-specified SIL programs into Simple programs and use Sample to infer additional specifications. Next, we want to feed these specifications back to the SIL verifier Silicon, hoping that they lead to a successful verification. In the following, we describe a sequence of steps to handle increasingly complex SIL programs.

### 2.1. Translate SIL Programs to Simple, Ignoring Permissions

In SIL, a method may only access a heap location if it has the appropriate permission. Specifications describe the transfer of permissions within a SIL program. For instance, methods use preconditions to specify which permissions they require from their caller. Postconditions indicate which permissions the methods return to their caller.

Currently, there is no semantics for any language constructs that involve permissions built into Sample. Thus, as a first step, we want to implement a translation of SIL ASTs to a Simple ASTs, ignoring any existing specifications that involve permissions. Analyzing the resulting Simple program is still meaningful, because the translation preserves the functional part of the program.

Some SIL constructs will require special attention: Sample currently does not consider method pre- and postconditions and does not support methods with multiple return values, as well as **`old`** expressions.

To test the correctness of the implementation, we can add support for the **`assert`** statement and check that Sample yields the expected result on SIL test programs.

Figure 2: Two possible heap abstractions for parameters `a` and `b` of method `m`.

### 2.2. Infer New Specifications with Sample

The next objective is to use Sample to infer new specifications for the SIL program and extend the program with these new specifications. We will then use the verifier again, trying to verify the extended SIL program.

Sample's new heap analysis [6] will be the main source of new specifications. The analysis represents an abstract heap as a labeled, directed graph, where nodes represent abstractions of objects. It distinguishes between definite nodes, representing exactly one object at runtime, and summary nodes, representing possibly many objects.

An edge between two nodes is annotated with a field and indicates that objects represented by the source node may point to objects represented by the target node via that field. In addition, each edge is annotated with the condition under which that edge exists. Edge conditions are often defined on value fields of the source and target nodes.

We will infer new specifications from abstract heaps. Initially, we will focus on non-recursive specifications such as permissions to individual fields and numerical constraints. Later, we will turn our attention to recursive data structures (see Sec. 2.4), which is exactly what Sample's heap analysis was designed to handle well.

### 2.3. Infer Heap Properties From Permissions

Existing SIL specifications that involve permissions are a valuable source of information about the heap of the program, in particular w.r.t. non-aliasing information.

#### 2.3.1. Example

Consider the following SIL method `m` with two parameters `a` and `b`.

```
method m(a: Ref, b: Ref)
  requires acc(a.f, write) && acc(b.f, write) { ... }
```

If we simply ignored the precondition, parameters `a` and `b` could point to the same object. As shown in Fig. 2 on the left, Sample's heap analysis would use a single summary node to approximate the method's parameters.

However, the method requires write permission to the field `f` of each of the objects referenced by `a` and `b`. SIL's permission model does not allow holding a write permission to the same object twice. Thus, the precondition implies that `a` and `b` in fact point to two separate objects. Telling Sample that `a != b` would result in a more precise heap abstraction (see Fig. 2).

### 2.3.2. Inference with Silicon

In our work, we do not plan to implement a sophisticated analysis for tracking permissions in Sample. Instead, we want to rely on the symbolic-execution-based verifier Silicon for handling permission-related reasoning. Even when Silicon fails to verify a program, it might have inferred useful heap properties such as the object disjointness in the above example.

We have chosen Silicon over Carbon as we can access the states that Silicon holds in all intermediate steps of the symbolic execution. We will extract from these states the heap properties inferred by Silicon and then explicitly add these properties to the SIL program in the form of additional method pre- and postconditions, loop invariants or `inhale` statements. In the above example, the extended method will have a new precondition `a != b`. Then, we will supply the extended program to Sample for analysis.

## 2.4. Support for Recursive Data Structures

Many interesting programs operate on recursive data structures such as linked lists. Recursive abstract predicates and abstraction functions are the key to specify such programs. An abstract predicate abstracts over heap locations and can express the permission to (and other properties of) a whole recursive data structure.

### 2.4.1. Example

Listing 1 in Appendix A contains the fully specified method `firstNats` that builds a linked list of the first $n$ natural numbers in ascending order.

`firstNats` uses the abstract predicate `valid` in its postcondition to express that it returns the permissions to the entire new list to the caller. The method also uses the abstraction function `sorted` to indicate that the returned list is sorted. Both `valid` and `sorted` are recursive. Given a reference `this` to an element of a list, they recurse to the next element `this.next` unless `this.next` is `null`, which marks the end of the list. A successful verification requires quite an elaborate loop invariant.

Without any annotations, Sample's heap analysis [6] infers the heap abstraction in Fig. 3 from the code. For example, the constraint $[(trg, \texttt{val}) = 0]$ tells us that `first` points to a list element whose value is 0. In this constraint, $trg$ represents an *edge-local identifier*. It refers only to the object pointed to by `first`, not every object represented by the summary node $\overline{n}_0$.

### 2.4.2. Relating Recursive Definitions to Heap Graphs with Summary Nodes

The abstract heap produced by Sample in the above example is closely related to the recursive definitions `valid` and `sorted`. The summary node has an edge to `null` and a self-loop, while the recursive abstract predicate `valid` also distinguishes between `next` being `null` and satisfying `valid` itself. Similarly, the recursive abstraction function `sorted` relates the values of two consecutive elements, just like the edge-local identifiers in the abstract value states associated with edges of the abstract heap.

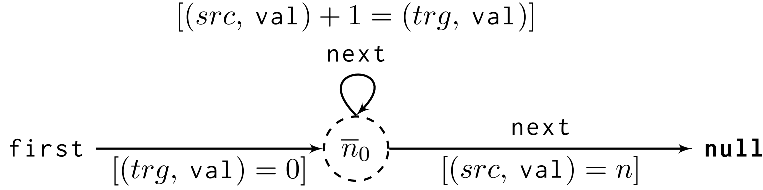


Figure 3: Heap abstraction at the end of the `firstNats` method.

A goal of this project is to exploit this connection between abstract heaps and abstract predicates in both directions of communication between Silicon and Sample. Firstly, it would be desirable to consider an abstract predicate that already exists in the program and build the corresponding abstract heap. Directly supplying that abstract heap to Sample's heap analysis could make the analysis more precise. Secondly, we want to infer new abstract predicates from the abstract heaps that Sample produces and feed these abstract predicates back to the verifier as part of the inferred specifications.

There is a mismatch between abstract heaps and abstract predicates in the sense that abstract predicates are fundamentally acyclic, which does not necessarily apply to abstract heaps with summary nodes. Constraints on edges may imply acyclicity, though.

# 3. Extensions

Depending on the available time and new insights gained during the project, there are several opportunities for further work:

**Reusing Existing Abstract Predicates in Inferred Specifications** As mentioned earlier, we plan to use existing abstract predicates to make the heap analysis more precise. In addition, it would make sense to reuse these abstract predicates in the inferred specifications rather than creating new ones.

This extension will require some form of check whether an abstract heap and abstract predicate match. We expect that reusing abstraction functions in inferred specifications will be even more challenging.

**Inferring Predicate Folding and Unfolding** An abstract predicate has a name and a possibly recursive body that defines its meaning. SIL verifiers distinguish between holding a predicate instance via its name and holding its body. **`fold`** and **`unfold`** statements tell the verifier to switch between one and the other.

As we plan to infer abstract predicates from the abstract heap, it is also desirable to learn where to put **`fold`** and and **`unfold`** statements. There seems to be a connection between merging and materializing heap nodes and the placement of **`fold`** and **`unfold`** statements, respectively.

**Integration of Permission Inference** Sample provides an analysis for inferring access permissions using linear programming [3]. This analysis could be exploited to reduce the annotation overhead for verifying SIL programs.

**Iterative Specification Refinement** It may be beneficial to iteratively exchange information between Sample's analysis and the SIL verifier.

Specifications inferred by Sample may not directly lead to a successful verification. However, they may at least enable Silicon to verify more parts of the program and infer additional information that may in turn make Sample's analysis more precise.

**Support for Complex SIL Types** Sample contains analyses for arrays and collections. Depending on how advanced these analyses are, we may use them to support SIL's dedicated types for sequences, sets and multisets.

**Magic Wand Operator** Recursive methods that operate on existing data structures usually unfold abstract predicates before a recursive call and fold them again afterwards. That is, permissions to the already processed parts of the data structure are implicitly saved on the call stack.

Because of such `fold` statements, such methods are not tail-recursive and can thus not be trivially represented as loops. There is not place to put the `fold` statement in a loop and thus, the loop loses the permissions to the parts of the data structure processed in earlier loop iterations.

A future SIL extension may introduce the magic wand operator $-\!\!*$ from separation logic [10], making it possible to specify loops in a more natural way. We may look into supporting this operator in specification inference.

# 4. Organization

## 4.1. Project Phases

The start of the project is on October 7th, 2013 and it ends on April 6th, 2014. The project consists of the following phases:

### 4.1.1. Ramp-up Phase

**Begin** October 7th, 2013

**End** November 4th, 2013

**Deliverables** Project proposal, initial presentation

**Description** The purpose of this phase is to collect and read papers relevant to the project, set up the development environment, learn the semantics of SIL and Simple and get an overview of the codebases. Discussions with the supervisors and the initial author of Sample will help determine the initial direction of the project.

Furthermore, preparing a range of SIL programs and manual translations to Simple will help to get a better understanding of the problem and serve as valuable testing artifacts in later phases.

#### 4.1.2. Conceptual Work and Implementation Phase

**Begin** October 28th, 2013

**End** March 7th, 2014

**Deliverables** Interim presentation

**Description** Sec. 2 gives an overview of the core objectives and a broad idea of how to approach them. Once they have largely been achieved, the focus will shift to the extensions as defined in Sec. 3.

#### 4.1.3. Writing Phase

**Begin** February 7th, 2014

**End** April 7th, 2014

**Deliverables** Project report, final source code, final presentation

**Description** The project report will describe in detail the conceptual and technical work done during the project. Writing should ideally begin two months before the end of the project, gradually replacing the implementation effort. The last three weeks should be devoted mostly to writing.

To facilitate writing, as many written notes and discussion summaries as possible will be collected throughout the earlier phases.

### 4.2. Presentations

Over the course of the project, there will be three presentations.

**Initial Presentation** The purpose of this 10-minute presentation is to inform the research group members about the project and gather initial feedback. It will take place roughly one month after the project start.

**Interim Presentation** Three months in, a 20-minute presentation discusses the current state of the project, existing problems and future work.

**Final Presentation** At the end of the project, a final presentation of 30 minutes gives an overview of has been achieved during the project and an outlook on future work. This is the only presentation that counts towards the grade.

### 4.3. Meetings

A meeting with both supervisors will take place weekly to discuss the project progress, possible problems and the next steps.

## References


[1] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.

[2] Bernhard Brodowsky. Translating Scala to SIL. Master's thesis, ETH Zurich, 2013.

[3] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. Static analysis of string values. In *Proceedings of the 13th international conference on Formal methods and software engineering*, ICFEM'11, pages 505–521. Springer, 2011.

[4] Pietro Ferrara. Static type analysis of pattern matching by abstract interpretation. In *Proceedings of the 12th IFIP WG 6.1 international conference and 30th IFIP WG 6.1 international conference on Formal Techniques for Distributed Systems*, FMOODS'10/FORTE'10, pages 186–200. Springer, 2010.

[5] Pietro Ferrara and Peter Müller. Automatic inference of access permissions. In *Proceedings of the 13th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, pages 202–218. Springer, 2012.

[6] Pietro Ferrara, Milos Novacek, and Peter Müller. Automatic inference of heap properties exploiting value domains. 2013.

[7] Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions.

[8] Christian Klauser. Translating Chalice into SIL. Bachelor's thesis, ETH Zurich, 2012.

[9] K. Rustan Leino and Peter Müller. A basis for verifying multi-threaded programs. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 378–393, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] Peter OHearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer Berlin Heidelberg, 2001.

[11] Malte Schwerhoff. Symbolic execution for Chalice. Master's thesis, ETH Zurich, 2010.

[12] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 148–172. Springer, 2009.

# A. Appendix

```
var next: Ref
var val: Int

predicate valid(this: Ref) {
  acc(this.next, write) && acc(this.val, write) &&
  (this.next != null ==> acc(this.next.valid(), write))
}

function sorted(this: Ref): Bool
  requires acc(this.valid(), epsilon)
{
  unfolding acc(this.valid(), epsilon) in (
    this.next != null ==> (
      this.val < (unfolding acc(this.next.valid(), epsilon)
      in this.next.val) && sorted(this.next))
  )
}

method firstNats(n: Int) returns (first: Ref)
  requires n >= 0
  ensures acc(first.valid(), write) && sorted(first)
{
  var tmp: Ref
  var i: Int

  tmp := null
  first := null
  i := n

  while (i >= 0)
    invariant
      ((i < n) ==> (first != null)) &&
      ((first != null) ==> (
           (acc(first.valid(), write))
        && (unfolding acc(first.valid(), write) in first.val == i + 1)
        && (sorted(first))))
  {
    tmp := new()
    tmp.val := i
    tmp.next := first
    i := i - 1
    first := tmp
    fold acc(first.valid(), write)
  }
}
```

Listing 1: Fully specified method that builds a list of the first $n$ natural numbers.