ETH zürich

ETH Library

Viper
A Verification Infrastructure for Permission-Based Reasoning

Report

Author(s):
Juhasz, Uri; Kassios, Ioannis T.; Müller, Peter; Nováček, Miloš; Schwerhoff, Malte; Summers, Alexander J.

Publication date:
2014

Permanent link:
https://doi.org/10.3929/ethz-a-010151765

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Viper: A Verification Infrastructure for Permission-Based Reasoning

Uri Juhasz, Ioannis T. Kassios, Peter Müller, Milos Novacek, Malte Schwerhoff, and Alexander J. Summers

Department of Computer Science, ETH Zurich, Switzerland
{uri.juhasz,ioannis.kassios,peter.mueller, milos.novacek,malte.schwerhoff,alexander.summers}@inf.ethz.ch

Abstract. The automation of verification techniques based on first-order logic specifications has benefited greatly from verification infrastructures such as Boogie and Why. These offer an intermediate language that can express diverse language features and verification techniques, as well as back-end tools such as verification condition generators.
However, these infrastructures are not well suited for verification techniques based on separation logic and other permission logics, because they do not provide direct support for permissions and because existing tools for these logics often prefer symbolic execution over verification condition generation. Consequently, tool support for these logics is typically developed independently for each technique, dramatically increasing the burden of developing automatic tools for permission-based verification.
In this paper, we present a verification infrastructure whose intermediate language supports an expressive permission model natively. We provide tool support, including two back-end verifiers, one based on symbolic execution, and one on verification condition generation; this facilitates experimenting with the two prevailing techniques in automated verification. Various existing verification techniques can be implemented via this infrastructure, alleviating much of the burden of building permission-based verifiers, and allowing the developers of higher-level techniques to focus their efforts at the appropriate level of abstraction.

1 Introduction

Over the last 15 years, static program verification has made significant progress. Among the theoretical and practical achievements which enabled this progress, two have been particularly influential. First, the development of widely-used common architectures for program verification tools, simplifying the development of new verifiers. Second, the development of permission logics (of which separation logic [28] is the most prominent example), simplifying the specification and verification of heap-manipulating programs and concurrent programs.

Many modern program verifiers use an architecture in which a front-end tool translates the program to be verified, together with its specification, into a simpler intermediate language such as Boogie [18] or Why [8]. The intermediate language provides a medium in which diverse high-level language features

and verification problems can be encoded, while allowing for the development of efficient common back-end tools such as verification condition generators. Developing a verifier for a new language or a new verification technique is, thus, often reduced to developing an encoding into one of these intermediate languages. For instance, Boogie is at the core of verifiers such as Chalice [22], Corral [15], Dafny [19], Spec# [21], and VCC [6], while Why powers for instance Frama-C and Krakatoa [7].

However, this infrastructure is generally not ideal for verifiers based on permission logics, such as separation logic. Verification condition generators and automatic theorem provers support first-order logic, but typically have no support for permission logics because of their higher-order nature. Therefore, most verifiers based on these specialised logics implement their own reasoning engines for each technique independently, which increases the burden of developing general-purpose automatic tools for permission-based verification.

In this paper, we present a verification infrastructure whose intermediate language Silver has a flexible notion of permissions built into its design. This allows for simple encodings of permission-based verification techniques. Along with the implementation of the language, we provide two back-end verifiers: Silicon, based on symbolic execution, and Carbon, based on verification condition generation (via an encoding into Boogie). Various existing approaches can be implemented via this common tool infrastructure, alleviating much of the burden of building permission-based verifiers, and allowing the developers of higher-level techniques to focus their efforts at this level of abstraction. The presented infrastructure enables research groups to build research prototypes more rapidly and facilitates the experimental evaluation and comparison of their results.

Outline. The next section describes the intermediate language Silver, and explains how various important high-level features can be encoded into the language. An example translation of a source program into Silver is presented in Section 3. Section 4 describes the back-end verifiers Silicon and Carbon. Section 5 discusses related work, and Section 6 concludes and outlines future work.

2 The Silver Language

The Silver language is designed with two equally-important goals in mind: first, to provide support for a variety of permission-based verification approaches and second, to facilitate the development of a variety of back-end tools such as verifiers based on symbolic execution and verification condition generation, as well as inference tools based on, for instance, abstract interpretation.

While Silver includes some basic programming features such as methods, loops, and conditionals, the emphasis of Silver is to provide a small core of verification primitives that allow one to encode a wide variety of source-level programming constructs and verification approaches. Permission-based verification approaches such as separation logics and implicit dynamic frames [29] control access to the program heap using some notion of permission. Permissions simplify framing (that is, proving that an assertion is not affected by a

heap modification), as well as reasoning about concurrency. *Fractional permissions* [4] allow for a permission to be split into parts and shared amongst several program entities (e.g. threads), although such a partial permission only permits reading from the corresponding heap location. The full permission is needed to write a heap location. In particular, so long as a method holds some permission to a memory location, no other method or thread can possibly be permitted to write to the location and, thus, its value can be assumed to remain stable.

The essence of many verification problems in permission-based approaches can be expressed generically in terms of when permission to certain heap locations is gained (along with information about the values of those locations), and lost (incurring a corresponding loss of information). These concepts can be expressed directly in Silver in the form of special `inhale` and `exhale` operations (see Section 2.2), which can be seen as permission-aware analogues of the `assume` and `assert` statements classically used to define verification conditions.

In this section, we describe the most important language constructs in Silver and some examples of their usage to implement important source-level features. For a more exhaustive description of the language, we provide a grammar in Appendix A. A test suite of example programs is also available along with our tools online, at `www.pm.inf.ethz.ch/research/viper`

2.1 Program Organisation

A Silver program consists of a sequence of global declarations, which can declare fields, methods, predicates, functions, and custom domains (see Section 2.5). Methods have no implicit receiver ("`this`" in many languages), but can have any number of in and out parameters. Predicates [24] can be used both to abstract over concrete assertions and to write recursive specifications of heap data structures. Functions can be used both in program expressions and in assertions, in a similar way to the use of pure methods in specification languages such as JML [16] and Spec# [1]. Function bodies are expressions rather than statements; in Silver this guarantees their evaluation to be side-effect free.

Verification of Silver programs is method-modular; method calls are verified with respect to the specification of the callee method, but not its implementation. The statements of the language do not introduce any concurrency explicitly. Instead, concurrent programming features must be encoded appropriately by a front-end tool; Section 3 shows this for a concrete example.

Program State. A Silver program state includes the current heap and the current assignment of values to program variables. In addition to the *current* heap, the meaning of "`old`" expressions can depend on the heap as it was in the prestate of the current method. Unlike in other intermediate verification languages such as Boogie, the heap is a "built-in" notion: it need not (and cannot) be manually defined and explicitly named in the Silver program. This design decision simplifies the static analysis of Silver programs via abstract interpretation, and also makes Silver programs easier to read and write.

The program state also tracks the permissions held by the current method invocation: to which heap locations permission is held, and how much. For debugging purposes, the current permissions to a particular field location can be queried by an expression `perm(e.f)`, although it is rarely necessary to do this. Instead, permissions are typically manipulated through statements that inhale or exhale permissions (see Section 2.2). Verifiers for Silver will check at various points that appropriate permissions are currently held. For example, when encountering an assignment `x:=y.f` (or indeed, any dereference of a heap location `y.f`), the Silver semantics requires that `y` is non-null and that a non-zero amount of permission is held to the heap location denoted by `y.f`. When modifying the heap location, the full permission to the location is required, as is standard when employing fractional permissions. These conditions are part of the Silver semantics and, thus, need not be written manually in a Silver program.

2.2 Permissions

Silver provides a number of constructs for writing specifications which prescribe a transfer of permissions. These constructs are based on implicit dynamic frames [29,20]. The most fundamental construct is the *accessibility predicate*, $\texttt{acc}(e_1.f, e_2)$, which represents permission to a single field location: the field f of the reference denoted by e_1. The optional expression e_2 specifies the *amount* of permission; by default, the full (write) permission is used. Silver supports syntax for fractional permissions [4], including basic arithmetic operators.

As an example, a method precondition `acc(x.f) && x.f > 0` expresses that the method requires the permission to the field location `x.f` and expects the value of this heap location to be positive[1]. An expression that dereferences the heap, such as `x.f` in the above example, will be checked to occur only in contexts where some permission to the corresponding heap location is held; in a method precondition, this can only be guaranteed if the permission is also required in the precondition, as in our example.

Inhale and Exhale. Just as first-order verification languages are typically based around `assume` and `assert` statements, verification in Silver centres around their permission-aware analogues: `inhale` and `exhale` statements. Conceptually, the operation `inhale` A acts just as an `assume` statement for all *pure* assertions (that is, assertions that do not include accessibility predicates); any permissions specified in A via accessibility predicates are *added* to the current program state. Dually, an `exhale` statement causes all first-order assertions to be asserted to be true, and all permissions to be *removed* from the current program state.

These basic operations can be used to define the verification semantics of many higher-level constructs. For example, a method call to a method whose

[1] In separation logic, this precondition would typically be written $\texttt{x.f} \mapsto v$ `&&` v `> 0` using an extra logical variable v; encoding separation logic into Silver is simple, and the formal relationship has been explored [25].

precondition is some assertion `pre` and postcondition is some assertion `post`, can be desugared into the two operations:

```
exhale pre
inhale post
```

Fork and join operations of a front-end language can be encoded in Silver analogously to a method call: a fork of a new thread is an `exhale` of the appropriate method's precondition, and a join is an `inhale` of the appropriate postcondition (of course, these need not occur immediately after one another in the program). Similarly (modulo considerations of deadlock avoidance), acquiring a monitor can be modelled as an `inhale` of the monitor invariant, and releasing can be modelled by a corresponding `exhale` [20]. This illustrates the flexibility of working at the level of `inhale` and `exhale` operations: details of the front-end language's runtime behaviour can be abstracted away; only the relevant proof obligations and permission manipulations need be represented in the Silver program.

Just as with `assume` statements in other verification languages, inserting arbitrary `inhale` statements into a Silver program can make it meaningless, from a verification perspective. For example, `inhale false` causes any subsequent proof obligations to be made trivial. It is the responsibility of the Silver writer (or front-end translation) to employ `inhale` statements only where they are justified, for instance, by a corresponding `exhale` somewhere else in the program.

For debugging purposes, Silver also provides an `assert` statement, which acts as an `exhale` but does not remove any permissions.

2.3 Recursive Definitions

Silver provides two main features for using recursive definitions in specifications: predicates (which can recursively define assertions), and functions (which can recursively define expressions for use in both specifications and implementations).

Predicates, Fold and Unfold. In order to specify permission to an unbounded number of heap locations, Silver assertions can include recursive predicates [24]. A predicate definition has a name, any number of parameters, and an assertion as its body. For example, the following predicate definition requires permissions to all locations of a linked list:

```
predicate list(l: Ref) {
  acc(l.val) && acc(l.next) &&
  (l.next ≠ null ⇒ list(l.next))
}
```

In a static verification tool, such a definition could be expanded an unbounded number of times, since the depth of recursion is not statically known, in general. In order to avoid the problem of potentially infinite unrollings of recursive predicate definitions, Silver takes the common approach that a predicate *instance* is *not* treated as purely synonymous with the assertion defined by its body [30].

Instead, a predicate instance such as `list(l)` can be explicitly exchanged for its body: a statement `unfold list(l)` first checks that the predicate instance is indeed held, removes that instance from the current state, and *inhales* the assertion from the predicate's body. For instance, `list(l)` needs to be unfolded in order to gain permission to `l.next`. A `fold` statement performs the inverse operation, exchanging a predicate's body for a predicate instance. The following example illustrates the use of `fold` and `unfold` in the traversal of a linked list:

```
method sum(l: Ref) returns (s: Int)
  requires list(l)
  ensures list(l)
{
  s := 0
  unfold list(l)
  if(l.next ≠ null) { s := sum(l.next) }
  s := s + l.val
  fold list(l)
}
```

Predicate definitions are interpreted according to their least fix-points; a built-in assumption in the verification of Silver is that a predicate instance never has an "infinite unrolling". This does not prevent predicates from describing cyclic data structures; their definitions could detect the cycle and prevent further recursion.

Note that although the verifiers provided for Silver (see Section Section 4) require explicit `fold` and `unfold` statements, these could potentially be inferred by a front-end tool or other static analysis.

2.4 Functions

As a complementary feature to predicates, Silver also provides recursive *functions*, to be used in specifications and implementations. A function definition consists of a function name, parameters, a precondition, an optional postcondition, and an expression for the function's body. The precondition must include sufficient permissions to allow all heap dereferences performed in the function's body. For example, the following function computes whether all elements of a linked-list data structure (as defined by the `list` predicate above) are greater than the specified parameter:

```
  function all_larger(l: Ref, n: Int): Bool
    requires list(l)
  { unfolding list(l) in
     l.val > n && (l.next ≠ null ⇒ all_larger(l.next, n) }
```

The `unfolding` expression has no effect on the value of the nested expression (after the "`in`"); its purpose is to temporarily unfold the specified predicate instance before checking whether appropriate permissions are held for the nested expression. In this example, this allows the nested expression to access `l.val` and `l.next`. An invocation of a function is allowed to occur only in contexts in which its precondition holds, for instance, in a method precondition such as:

```
list(l) && all_larger(l,0)
```

Function definitions must guarantee termination in states in which their preconditions hold. This termination check may depend on the finite-unrolling of predicate instances; in the example above, the fact that a predicate instance is unfolded around the recursive call is sufficient to pass termination checks[2].

Note that the constraints imposed by including the function in the example assertion above could instead have been included in the definition of the `list` predicate, removing the direct need for a function definition. However, this would make the `list` predicate no longer reusable in other contexts in which different constraints on the list values (such as sortedness) might alternatively be required. Furthermore, functions can be used also in implementations (their use is similar to that of pure methods in specification languages such as JML [16]).

2.5 Types and Domains

Silver includes a very simple type system. The built-in primitive types are: `Int`, `Bool`, `Ref` and `Perm` (for expressions denoting permission amounts). The type `Ref` encompasses all references in the program; any more fine-grained notion of typing (in particular, class types) must be encoded by front-end tools. Silver also provides built-in support for polymorphic sequence and set types, written `Seq[T]` and `Set[T]`, respectively. Standard operations on sequences and sets are pre-defined; support in the verifiers is achieved via an axiomatisation based on that used in Dafny [17].

In addition to the built-in types, custom *domain types* can be defined, including appropriate axioms and functions about the corresponding domain types. Declarations of domain types may be polymorphic (via type parameters). They are global to the program, and cannot refer to a program state. As an example of a custom domain type, one can easily define generic (fixed-length) tuple types, such as pairs:

```
domain Pair[X,Y] {
  function pair(x: X, y: Y) : Pair[X,Y]
  function first(p: Pair[X,Y]) : X
  function second(p: Pair[X,Y]) : Y

  axiom ax1 { (∀ x:X, y:Y • first(pair(x, y)) == x) }
  axiom ax2 { (∀ x:X, y:Y • second(pair(x, y)) == y) }
}
```

Note that domain types are not ADTs; built-in support for constructors and induction may be added in future versions of Silver.

2.6 Paired Assertions

When encoding a high-level verification approach into an intermediate language, it is often appropriate to allow some properties to be used without checking them; these properties need to be justified elsewhere, for instance, by a soundness proof

[2] At present, this is the only termination measure supported in Silver, but we plan to extend this to more general termination measures in the near future.

for a verification technique. Other intermediate languages express such properties via `assume` statements and designated specification constructs such as *free* pre and postconditions [18].

Silver provides a uniform way of adding externally-justified properties to any assertion (such as method preconditions, loop invariants, and predicate definitions) via *paired assertions* of the form [A_1,A_2]. Inhaling such as assertion means the same as inhaling A_1, while exhaling means exhaling A_2. For instance, Boogie's free precondition is expressed in Silver as a regular precondition with a paired assertion whose second component is simply `true`.

One application of paired assertions is to provide the verifier (which does not understand induction natively) with the extra flexibility that an instance of inductive reasoning would provide. For example, the following paired assertion allows the verifier to use the more direct form of a quantified assertion when inhaling, but to prove instead the appropriate premise of a proof by (strong) induction when exhaling:

```
[∀ x: Int • x ≥ 0 ⇒ P(x) ,
 ∀ x: Int • (∀ y: Int • y≥0 && y<x ⇒ P(y)) && x ≥ 0 ⇒ P(x)
]
```

2.7 Wildcard and Abstract Read Permissions

Fractional permissions allow one to distinguish between read and write access to heap locations, but can lead to specification overhead, due to having to select appropriate concrete fractions when writing specifications. For most read accesses, it is sufficient to check that *some* positive amount of permission is held, but the exact amount is irrelevant. For this purpose, Silver allows a syntax `acc(e.f,wildcard)`, which represents *some* positive permission amount. When exhaling this assertion, provided some positive amount of permission is held, the amount given away is always strictly smaller, and need not be specified concretely.

A drawback of `wildcard` permissions is that once it has been given away, it is not possible to re-gain the same permission amount, for instance, to reobtain a full permission. That is, giving away a `wildcard` permission to a location make the location immutable. In scenarios in which a read permission needs to be given away, but later re-obtained, the need to specify concrete fractions can still be avoided through the use of *abstract read permissions* [11]. This idea allows programs to introduce a name for some fresh permission amount which, during a following sequence of exhales will always be guaranteed to be smaller than the amounts currently held. By giving a name to the amount, it can be tracked precisely in specifications. By guaranteeing that the exhaled amount is strictly smaller than the amounts held, some amount of permissions also always remains after the exhales, which simplifies framing.

In Silver, the statement `fresh k;` (for some variable `k` of type `Perm`) assigns an arbitrary, positive permission amount to `k`. This amount is constrained further during block statements of the form `constraining (k) { ... }` . Executing the

body of the block (usually consisting of one or more `exhale` statements) adds extra constraints on the (under-specified) value of `k`, to reflect it being chosen to be "small enough". A methodology for soundly exploiting this functionality for method calls and fork/join concurrency is described in detail in [11].

3 Using Silver: An Example

Figure 1 presents the famous Owicki-Gries example, in which two concurrently-running threads compete for a lock, each wishing to increment the same shared counter. The Owicki-Gries specification approach is to employ two extra *ghost variables* (represented as extra fields `g1` and `g2` in the example code), representing each thread's contribution to the value stored in `count`. The permission to access these ghost fields is split between the monitor invariant (which can be obtained only by acquiring the monitor), and the respective threads. Since the monitor invariant maintains a constraint about the `count` variable, while the individual threads can talk about their own ghost variables, the combined effect of the threads can be reasoned about in the `OwickiGries` method.

```
0   class OG {
1     var count : int;
2     ghost var g0, g1 : int;
3
4     monitor-invariant :=
5     acc(count) && acc(g0, 1/2) && acc(g1, 1/2) && count == g0 + g1
6
7     method OwickiGries() {
8       var x := new OG{ count := 0, g0 := 0, g1 := 0 }
9       share x // create monitor
10
11      fork worker0 := x.Worker(true)
12      fork worker1 := x.Worker(false)
13      join worker0; join worker1
14
15      acquire x
16      assert x.count == 2
17    }
18
19    method Worker(b: bool)
20      requires (b ? acc(g0, 1/2) : acc(g1, 1/2))
21      ensures b ⇒ acc(g0, 1/2) && g0 == old(g0) + 1
22      ensures !b ⇒ acc(g1, 1/2) && g1 == old(g1) + 1
23    {
24      acquire this
25      count := count + 1
26      if (b) { g0 := g0 + 1 }
27      else   { g1 := g1 + 1 }
28      release this
29    }
30  }
```

Fig. 1. Owicki Gries example, at source level (based on syntax of Chalice language).

```
0  field count: Int;
1  field g0: Int; field g1: Int
2
3  method OwickiGries() {
4    var x : Ref;  x := new(*); // inhale permission to all fields
5    x.count := 0;  x.g0 := 0;  x.g1 := 0;
6
7    exhale acc(x.count) && acc(x.g0,1/2) && acc(x.g1,1/2)
8             && x.count == x.g0 + x.g1 // create monitor
9
10   var old_g0 : Int := x.g0; // save old(x.g0)
11   exhale acc(x.g0,1/2) // fork Worker(x,true)
12
13   var old_g1 : Int := x.g1 // save old(x.g1)
14   exhale acc(x.g1,1/2) // fork Worker(x,false)
15
16   // join threads
17   inhale acc(x.g0,1/2) && x.g0 == old_g0 + 1
18   inhale acc(x.g1,1/2) && x.g1 == old_g1 + 1
19
20   inhale acc(x.count) && acc(x.g0,1/2) && acc(x.g1,1/2)
21            && x.count == x.g0 + x.g1 // acquire monitor
22   assert x.count == 2
23 }
24
25 method Worker(this:Ref, b:Bool)
26   requires b ? acc(this.g0, 1/2) : acc(this.g1, 1/2)
27   ensures  b ⇒ acc(this.g0, 1/2) && this.g0 == old(this.g0) + 1
28   ensures !b ⇒ acc(this.g1, 1/2) && this.g1 == old(this.g1) + 1
29 {
30   inhale acc(this.count) && acc(this.g0, 1/2) && acc(this.g1, 1/2)
31            && this.count == this.g0 + this.g1 // acquire monitor
32
33   this.count := this.count + 1
34   if (b){  this.g0 := this.g0 + 1  }
35   else  {  this.g1 := this.g1 + 1  }
36
37   exhale acc(this.count) && acc(this.g0,1/2) && acc(this.g1,1/2)
38            && this.count == this.g0 + this.g1 // release monitor
39 }
```

Fig. 2. Owicki Gries example from Figure 1, encoded in Silver.

Figure 2 shows a possible encoding of the program of Figure 1 into Silver. Note that the concurrency constructs no longer occur explicitly in the program; instead, they have been replaced with appropriate `inhale` and `exhale` statements, capturing the same effect from a verification perspective. For example, acquiring a monitor is replaced by an `inhale` of the corresponding monitor invariant, while forking a thread is replaced by an `exhale` of its precondition. Note that the specifications of `Worker` have been simplified at call-site (based on the known value of the boolean parameter); such simplifications may or may not be performed by a front-end tool, and are performed in the Silver AST when they can be resolved statically. Note also that the encoding of `old` expressions from the source program requires recording the values of the corresponding ex-

pressions at the pre-states of the forked threads: exactly which expressions need their values saving can be computed based on the (simplified) specifications of `Worker`.

As discussed in Section 2, the Silver representation of the program can abstract over the concrete constructs used in the source program, and concentrate on their verification semantics. The Silver file shown verifies in both of our verifiers (Carbon and Silicon), which are described in the next section.

4 The Viper Infrastructure

The Viper verification infrastructure is built around Silver, as depicted in Figure 3. The Silver AST is constructed by a *front-end*. Currently, Viper includes three front-ends: a parser for textual Silver files, as well as two proof-of-concept translations: one for Chalice [22] and one for a small subset of Scala. Verification of Silver programs is performed by a *back-end*. Viper includes two stable back-ends: Silicon, a verifier based on symbolic execution (SE), and Carbon, a verifier based on verification condition generation (VCG). Both use the theorem prover Z3 [23] to discharge proof obligations; Silicon uses Z3 directly, while Carbon encodes Silver programs into Boogie programs.

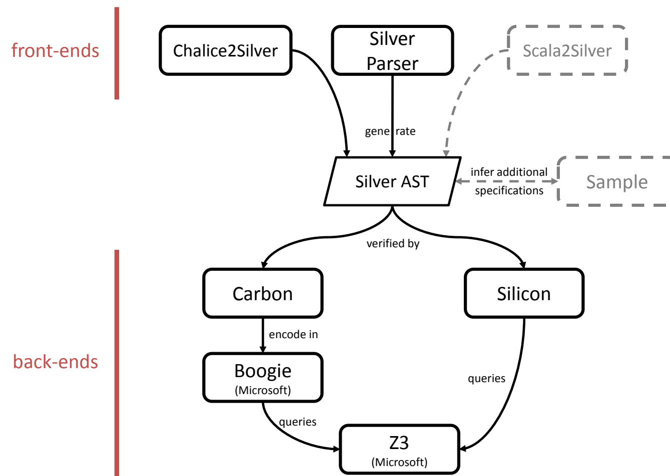

Fig. 3. The Viper infrastructure built around Silver. Rounded boxes denote tools; tools depicted as solid boxes are considered reasonably stable, whereas dashed boxes are in an experimental stage and not described in this paper.

All of our tools are implemented in Scala and can thus be used under Windows, Mac OS and Linux (Boogie and Z3 can also be compiled for these systems).

4.1 Verification Condition Generation vs. Symbolic Execution

One of the key differences between VCG and SE is that VCG encodes all relevant information about the method to be verified into a verification condition and lets the theorem prover reason about it, whereas SE for permission logics [2] reasons about heap properties directly in the verifier and queries the theorem prover only for non-heap-related properties such as arithmetic. Experiments previously conducted by us [13] have shown that SE tends to be faster than VCG, but they also showed that the performance gain comes at the cost of occasional incompletenesses in the verification. Having an SE-based and a VCG-based verifier for the same language (and using the same theorem prover) facilitates experimenting with the two prevailing techniques in automated verification, and helps us to develop a deeper understanding of which problems are more amenable to which technique. We have observed, for example, that VCG makes it easier to support elements of regular first-order logic such as quantifiers and (potentially heap-dependent) functions, whereas SE facilitates experimenting with variations and extensions of the notion of permissions used in the verifiers.

The next two subsections give a brief overview of how certain key problems are handled by Carbon using VCG and by Silicon using SE.

4.2 Verification Condition Generation with Carbon

In Carbon, each method and its specification are encoded as a single implication of the form $\text{Pre} \Rightarrow \text{WP}(\text{Body},\text{Post})$, stating that the given method precondition implies the weakest precondition of the method body w.r.t. the given method postcondition. This (often very large) implication is then given to the theorem prover.

Carbon's encoding of Silver programs into Boogie tracks information about heap values and currently-held permissions in two map-typed variables in the Boogie program. The heap H maps receiver-field pairs to values, whereas the permission mask M maps them to permission amounts. Intuitively, the locations to which M currently stores a positive permission amount are those that may be read from H at the current program point. For details of this encoding, see [20,10].

Silver's `inhale` and `exhale` operations manipulate the heap and permission mask. The `inhale` operation adds permissions to the mask and assumes properties of heap values; `exhale` asserts properties of heap values, checks that the required permissions are available, removes them, and assigns arbitrary values to all heap locations to which no permission is subsequently held. This last *havoc* step reflects that other methods may, at this point, have write permission to these locations and could modify their values.

4.3 Symbolic Execution with Silicon

As in similar tools for permission-logics [2,12], the symbolic state in Silicon consists of a symbolic heap and a set of path conditions. The symbolic heap

contains information about heap locations and permissions, whereas the path conditions (first-order facts) describe constraints about the values of variables and heap locations. Only the path conditions, but not the symbolic heap, are given to a theorem prover when a (non-heap-related) property must be checked.

The symbolic heap is a set of *heap chunks*, representing (only) those heap locations to which permission is currently held; all other heap locations are not a part of the symbolic heap. Each heap chunk maps a heap location to a value and a (positive) permission amount. Both the value and the permission amount are either statically-known constants or symbolic values. Properties of the latter are described via constraints in the path conditions.

Silver's `inhale` and `exhale` operations directly manipulate the symbolic heap. The `inhale` operation adds heap chunks with fresh symbolic values, and path conditions about the appropriate symbolic values. `exhale` checks properties of symbolic values (by asking the prover whether they are implied by the path conditions), and (for accessibility predicates) also checks that an appropriate heap chunk is present, and then removes it.

When using fractional permissions, one heap location may occur in several heap chunks. To determine the total amount of permission available for a heap location, Silicon sometimes needs to apply *heap compression*. That is, for each pair of heap chunks for the same field name, it asks the prover whether, from the set of path conditions, it can deduce that the two receiver objects are aliases. If so, the two heap chunks are merged into one, with the sum of the permission amounts, and an additional path condition is added stating that the values are the same.

5 Related Work

Two intermediate languages and verification condition generators that are widely used for the development of program verifiers are Boogie [18] and Why [8]. These verification condition generators have no direct support for permissions. Permission-related operations can be encoded, as is indeed achieved by both Chalice [20] and our verifier, Carbon. However, having direct support for permission-based reasoning greatly simplifies the development of front-ends and also enables the use of Smallfoot-style symbolic execution [2].

While Silver is similar to Boogie in some respects, several design decisions have been taken differently in order to facilitate specification *inference* via abstract interpretation. For instance, Silver has a built-in heap, heaps cannot be stored in variables, and heaps are indexed by references and *constant* field names. In particular, it is not possible to quantify over field names.

Recent work on decision procedures for separation logic [26,27] provides the potential to simplify the encoding of proof obligations for permission logics into SMT solvers. However, these decision procedures currently target restricted forms of separation logic, and do not support fractional permissions or user-defined predicates and functions.

coreStar [3] is an intermediate language and verifier for separation logic that includes a symbolic execution engine. Front-ends implemented on top of coreStar must define an encoding of the program to be verified into coreStar's language and also have to provide proof rules and abstraction rules to customize the behavior of the verifier and the inference. In comparison, our verification infrastructure is sufficiently expressive to capture a wide variety of languages and verification techniques, and does not require front-end developers to provide their own proof and abstraction rules. Furthermore, having a fixed language and rule set allows our verifiers to be more specialised.

Other automatic verifiers for separation logic, such as Smallfoot [2] and Veri Fast [12] do not share a common infrastructure (other than the SMT solver). Each of them implements their own verification engine based on symbolic execution.

6 Conclusions and Future Work

Viper is a verification infrastructure for building modular program verifiers based on permission logics. Its intermediate language Silver provides a flexible permission model, and supports user-defined predicates and functions. Viper includes the back-end verifiers Carbon (using verification condition generation) and Silicon (using symbolic execution). Supporting both verification condition generation and symbolic execution enables experiments with the two prevalent techniques for automatic verification. We additionally provide a parser for Silver, and two front-end translation tools for Chalice and a subset of Scala. The development of these translators helped us to fine-tune the design of the Silver language and to practically evaluate the developed infrastructure.

At the time of writing, several components of Viper are available online at `www.pm.inf.ethz.ch/research/viper`. An official release is planned for the early summer of this year. By that time, we plan to provide IDE support and a verification debugger; both components are currently under development. Our future work will focus on further increasing the expressiveness of the Silver language, as well as developing an inference component for Silver specifications based on our abstract interpreter, Sample.

Acknowledgements. This work has been funded in part by the Swiss National Science Foundation and by the Hasler Foundation. It has benefited from projects by Stefan Heule [9], Bernhard Brodowsky [5] and Christian Klauser [14].

References

1. M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. *Communications of the ACM*, 54(6):81–91, 2011.

2. J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, volume 4111 of *LNCS*, pages 115–137. Springer, 2006.
3. M. Botincan, D. Distefano, M. Dodds, R. Grigore, D. Naudziuniene, and M. J. Parkinson. coreStar: The core of jStar. In K. R. M. Leino and M. Moskal, editors, *BOOGIE*, pages 65–77, 2011. http://research.microsoft.com/en-us/um/people/moskal/boogie2011/boogie_2011_all.pdf.
4. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *SAS*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
5. B. Brodowsky. Translating Scala to SIL. Master's thesis, Dept. of Computer Science, ETH Zurich, 2013.
6. E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In T. Touili, B. Cook, and P. Jackson, editors, *CAV*, volume 6174 of *LNCS*, pages 480–494. Springer, 2010.
7. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.
8. J.-C. Filliâtre and A. Paskevich. Why3—where programs meet provers. In M. Felleisen and P. Gardner, editors, *ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.
9. S. Heule. Verification condition generation for the intermediate verification language SIL. Master's thesis, Dept. of Computer Science, ETH Zurich, 2013.
10. S. Heule, I. T. Kassios, P. Müller, and A. J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In G. Castagna, editor, *ECOOP*, volume 7920 of *LNCS*, pages 451–476. Springer, 2013.
11. S. Heule, K. R. M. Leino, P. Müller, and A. Summers. Abstract read permissions: Fractional permissions without the fractions. In *VMCAI*, LNCS. Springer, 2013.
12. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011.
13. I. T. Kassios, P. Müller, and M. Schwerhoff. Comparing verification condition generation with symbolic execution: An experience report. In *VSTTE*, pages 196–208, 2012.
14. C. Klauser. Translating Chalice into SIL. Bachelor's thesis, Dept. of Computer Science, ETH Zurich, 2012.
15. A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In P. Madhusudan and S. A. Seshia, editors, *CAV*, volume 7358 of *LNCS*, pages 427–443. Springer, 2012.
16. G. Leavens, A. L. Baker, and C. Ruby. JML: a notation for detailed design. In I. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.
17. K. R. M. Leino. Specification and verification of object-oriented software. In *Marktoberdorf International Summer School 2008, Lecture Notes*, 2008.
18. K. R. M. Leino. This is Boogie 2. Working draft; available at http://research.microsoft.com/en-us/um/people/leino/papers.html, 2008.
19. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *LPAR*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

20. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *ESOP*, volume 5502 of *LNCS*, pages 378–393. Springer, 2009.
21. K. R. M. Leino and P. Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In P. Müller, editor, *Advanced Lectures on Software Engineering—LASER Summer School 2007/2008*, volume 6029 of *LNCS*, pages 91–139. Springer, 2010.
22. K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705 of *LNCS*, pages 195–222. Springer, 2009.
23. L. Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
24. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.
25. M. Parkinson and A. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3), 2012.
26. J. A. N. Pérez and A. Rybalchenko. Separation logic modulo theories. In C. Shan, editor, *APLAS*, volume 8301 of *LNCS*, pages 90–106. Springer, 2013.
27. R. Piskac, T. Wies, and D. Zufferey. GRASShopper—complete heap verification with mixed specifications. In E. Ábrahám and K. Havelund, editors, *TACAS*, volume 8413 of *LNCS*, pages 124–139. Springer, 2014.
28. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
29. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In S. Drossopoulou, editor, *ECOOP*, volume 5653 of *LNCS*, pages 148–172. Springer, 2009.
30. A. Summers and S. Drossopoulou. A formal semantics for isorecursive and equirecursive state abstractions. In *ECOOP*, volume 7920 of *LNCS*, pages 129–153. Springer, 2013.

A The Silver syntax

This section describes the syntax of Silver in BNF. The grammar does not model the binding strength of operators. The special repetition operator ,* is used to denote a comma-separated list. Semicolons are optional after declarations and statements and not mentioned here for clarity.

```
// Declarations
sil-program ::=
  (
    domain     |
    field      |
    function   |
    predicate  |
    method
  )*

domain ::=
  "domain" domain-name "{"
    domain-function*
    axiom*
  "}"

domain-name ::=
  ident |
  ident "[" ident,*  "]"  //e.g. Seq[T]

domain-function ::=
  ["unique"] "function" ident function-signature

function-signature ::=
   formal-args ":" type

axiom ::=
  "axiom" ident "{" exp "}"

field ::=
  "field" ident ":" type

function ::=
  "function" ident function-signature
    precondition*
    postcondition*
  "{" exp "}"

precondition ::=
  "requires" exp
```

```
postcondition ::=
  "ensures" exp

invariant ::=
  "invariant" exp

predicate ::=
  "predicate" ident formal-args "{" exp "}"

method ::=
  "method" ident formal-args [formal-returns]
      precondition*  postcondition*
    block

formal-args ::=
  "(" formal-arg,*  ")"

formal-arg ::=
  ident ":" type

formal-returns ::=
  "returns" formal-args

// Statements
block ::=
  "{" stmt "}"

stmt ::=
  "var" ident  ":" type [":=" exp] // local variable
                                   // declaration with an
                                   // optional initial
                                        value
  ident          ":=" exp | // local variable assignment
  field-access   ":=" exp | // field assignment
  ident          ":=" "new(*)" | // object creation
                              // (all fields)
  ident          ":=" "new(" ident,*  ")" | // obj. creation
                                    // (specified fields)

  "assert" exp |
  "assume" exp |
  "inhale" exp |
  "exhale" exp |
  "fold"   acc-exp |
  "unfold" acc-exp |

  "goto" ident    |           // goto statement
  ident ":"       |           // a goto label

  if-statement    |
```

```
  while-statement |
  call-statement  |
  fresh-block

if-statement ::=
  "if" "(" exp ")"
    statement-block
  ("elsif" "(" exp ")"
    statement-block
  )*            // any number of elseif branches
  ["else"
    statement-block
  ]             // optional else branch

while-statement ::=
  "while" "(" exp ")"
      invariant*
    statement-block

call-statement ::= // method call [with return target]
  [ident,* :=] ident "(" exp,* ")"

fresh-statement ::=
  "fresh" ident,*

constraining-block ::=
  "constraining" "(" ident,* ")"
    block

// Expressions
binop ::=
  "==" | "!=" |                     // equality operators
  "==>" | "||" | "&&" |"<==>" |     // boolean operators
  "<" | "<=" | ">" | ">=" |         // ordering
                                    //  (integers and
                                       permissions)
  "+" | "-" | "*" |                 // arithmetic operators
                                    //  (integers and
                                       permissions)
                                    // also int*permission
  "\\" | "\%" |                     // arithmetic division
                                    //  and modulo
  "\/" |                            // permission division
                                    //  (of two integers)

  "union" | "intersection" | "setminus"  //set operators
  "++"  |           // sequence concatenation
  "in"  |           // set/multiset/sequence membership
  "subset"          // subset relation
```

```
unop ::=
  "!" |              // boolean negation
  "+" | "-"          // integer and permission

exp ::=
  "true"  | "false" |          // boolean literal
  integer |                    // integer literal

  "null"   |                   // null literal
  "result" |                   // result literal in
                               //  function postconditions
  ident    |                   // local variable read

  "(" exp ")" |

  unop exp |                  // unary expression
  exp binop exp |             // binary expression
  exp "?" exp ":" exp |       // conditional expression

  "old" "(" exp ")"            // old expression

  "none"     |              // no permission literal
  "write"    |              // full permission literal
  "epsilon"  |              // epsilon permission literal
  "wildcard" |              // wildcard permission
  exp "/" exp |             // concrete fractional
                            //  permission

  "perm" "(" loc-access ")" | // current permission
                              //  of given location

  acc-exp  |                   // accessibility predicate

  ident "(" exp,*  ")" |  // [domain] function
                          //  application
  field-access         |  // field read
  predicate-access     |  // predicate access

  "[" exp "," exp "]" |        // inhale exhale expression
  "unfolding" acc-exp in exp | // unfolding expression

  // quantification
  "forall" formal-arg,*  "::" trigger,*  exp |
  "exists" formal-arg,*  "::" trigger,*  exp |

  seq-constructor-exp      |
  set-constructor-exp      |

  seq-op-exp |
  set-op-exp
```

```
seq-constructor-exp ::=
  "Seq" "[" type "]" "()" | // the empty sequence
  "Seq" "(" exp,*  ")"     | // explicit sequence
  "[" exp ".." exp ")"       // half-open range of numbers

set-constructor-exp ::=
  "Set"      "[" type "]" "(" exp,*  ")" | // explicit set
  "Multiset" "[" type "]" "(" exp,*  ")"   // explicit
                                           //   multiset
set-op-exp ::=
  "|" exp "|"                 // set/multiset cardinality

seq-op-exp ::=
  exp "[" exp "]"           |  // sequence lookup
  exp "[" ".." exp "]"      |  // take n first elements
  exp "[" exp ".." "]"      |  // drop n last elements
  exp "[" exp ".." exp "]"  |  // take and drop
  exp "[" exp ":=" exp "]"  |  // update sequence at

  "|" exp "|"                  // length of a sequence

trigger ::=
  "{" exp,*  "}"  // a trigger for a quantification

acc-exp ::=
  "acc" "(" loc-access ["," exp ]")" //access
                                     // default is write
loc-access ::=
  field-access | predicate-access

field-access ::=
  exp "." ident |               // field access

predicate-access ::=
  ident "(" exp,*  ")"         // predicate access

// Types
type ::=
  "Int" | "Bool" | "Perm" | "Ref" |  // primitive types
  "Seq" "[" type "]"                 // sequence type
  "Set" "[" type "]"                 // set type
  "Multiset" "[" type "]"            // multiset type
  ident [ "[" type,*  "]" ]
               // [instance of a generic] domain type

// Identifiers
  ident ::=    // regular expression for an identifier
    "[a-zA-Z$_][a-zA-Z0-9$_']*"
```