# Viper: A Verification Infrastructure for Permission-Based Reasoning

Peter Müller, Malte Schwerhoff, and Alexander J. Summers

Department of Computer Science, ETH Zurich, Switzerland
{peter.mueller, malte.schwerhoff, alexander.summers}@inf.ethz.ch

**Abstract.** The automation of verification techniques based on first-order logic specifications has benefitted 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: in particular, verification condition generators.
However, these infrastructures are not well suited to 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 favour symbolic execution over verification condition generation. Consequently, tool support for these logics (where available) 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; an inference tool based on abstract interpretion is currently under development. A wide range of 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 reasoning techniques to focus their efforts at an appropriate level of abstraction.

## 1 Introduction

Over the last 15 years, static program verification has made wide-ranging and significant progress. Among the many theoretical and practical achievements that 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 [34] 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 [22] or Why [14]. 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 [26], Corral [20], Dafny [23], Spec# [25], and VCC [11], while Why powers for instance Frama-C [19] and Krakatoa [13].

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, typically based on symbolic execution, for each technique independently, increasing the burden of developing general-purpose automatic tools for permission-based verification.

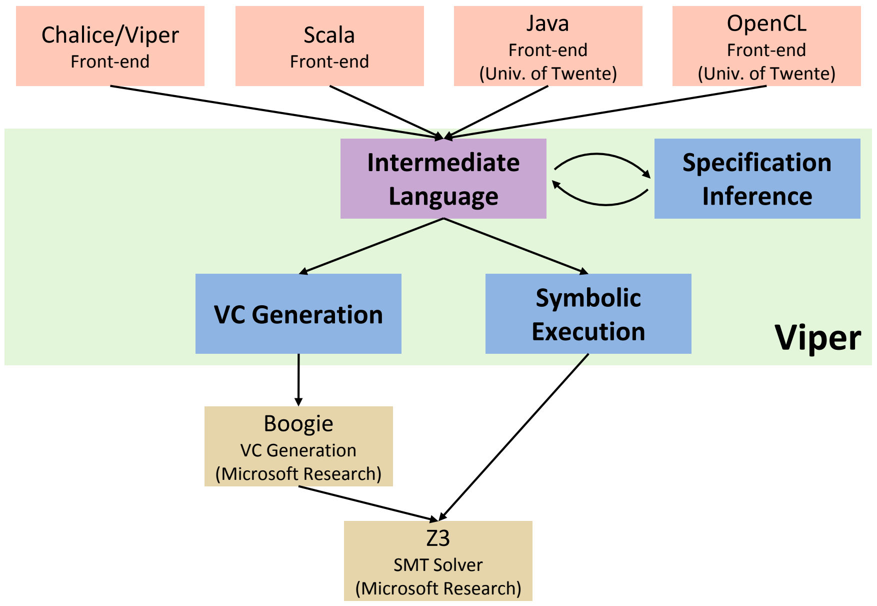

**Fig. 1.** The Viper infrastructure, underlying tools and currently-existing front-ends. All Viper components are implemented in Scala and can thus be used under Windows, Mac OS and Linux (Boogie and Z3 can also be compiled for these systems).

In this paper, we present Viper, a verification infrastructure whose intermediate language includes a flexible permission model, allowing for simple encodings of permission-based reasoning techniques. The Viper infrastructure provides two back-end verifiers, one using symbolic execution and one using verification condition (VC) generation (via an encoding into Boogie); a specification inference via abstract interpretation is under development. Currently, Viper is targeted by four front-end tools: we developed front-ends for a re-implementation of Chalice and for a small subset of Scala; front-ends for Java and for OpenCL [4] have been developed in the context of the VerCors project [5]. Several additional front-ends are under development. Fig. 1 gives an overview of the Viper infrastructure.

The Viper infrastructure serves three main purposes:

1. Viper facilitates the development of program verifiers based on permission logics, alleviating much of the involved burden by making a large portion of the tool chain reusable, and allowing the developers of higher-level techniques to focus their efforts at this level of abstraction. To support this purpose, Viper provides an expressive intermediate language with primitives that let front-ends encode a wide range of source languages, specifications, and verification techniques. Moreover, the Viper back-ends provide a high degree of automation, aiming to eliminate situations in which tool developers and users need to understand the internals of the back-ends in order to guide the verification effort. This automation is crucial to preserving both the abstractions provided by the Viper infrastructure and the front-ends developed on top of it.
2. Viper allows researchers to rapidly prototype and experiment with new verification techniques by encoding them manually in our intermediate language without (initially) developing a dedicated front-end. To support this purpose, Viper's intermediate language is human readable and provides high-level features such as methods and loops. A parser and type-checker allow one to write Viper code directly.
3. Viper supports the comparison and integration of different verification back-ends. To support this purpose, Viper provides two deductive verifiers and an abstract interpreter. The intermediate language is designed to cater for different reasoning techniques, for instance by providing a heap model similar to those of source languages (facilitating, for example, the use of existing heap analyses).

**Outline.** This paper gives an overview of the Viper intermediate language. The next section surveys key features of the language and illustrates how they are used to encode more abstract languages and verification techniques. The subsequent sections provide more details on permissions and predicates (Sec. 3), the specification of functional behaviour (Sec. 4), and the encoding of mathematical theories (Sec. 5). We present an evaluation in Sec. 6, summarise related work in Sec. 7, and conclude in Sec. 8. A comprehensive set of examples, including all examples presented in this paper, as well as manually encoded examples from verification competitions, is available in an online appendix [28].

## 2 Viper in a Nutshell

The Viper infrastructure is centred around a sequential, imperative, object-based intermediate language. A program in this language consists of a sequence of global declarations for fields, methods, functions, predicates, and custom domains. There is no notion of class; every object has every field declared in the program, and methods and functions have no implicit receiver. Predicates [30] can be used both to abstract over concrete assertions and to write recursive specifications of heap data structures. Custom domains are used to declare mathematical theories. Verification of Viper programs is method-modular; method calls are verified with respect to the specification of the callee and not its implementation.

In this section we illustrate the core features of the Viper language using two examples. We use an implementation of a sorted list to illustrate how Viper supports the specification and verification of heap data structures. We then use a client of the list to demonstrate how to encode language features and verification approaches which are not directly available in Viper.

### 2.1 Specification and Verification of Heap Data Structures

```
 1 field data: Seq[Int]
 2
 3 define sorted(s) forall i: Int, j: Int :: 0 <= i && i < j && j < |s|
 4                                     ==> s[i] <= s[j]
 5
 6 method insert(this: Ref, elem: Int) returns (idx: Int)
 7   requires acc(this.data) && sorted(this.data)
 8   ensures  acc(this.data) && sorted(this.data)
 9   ensures  0 <= idx && idx <= old(|this.data|)
10   ensures  this.data == old(this.data)[0..idx] ++
11                        Seq(elem) ++ old(this.data)[idx..]
12 {
13   idx := 0
14   while(idx < |this.data| && this.data[idx] < elem)
15     invariant acc(this.data, 1/2)
16     invariant 0 <= idx && idx <= |this.data|
17     invariant forall i: Int :: 0 <= i && i < idx
18                          ==> this.data[i] < elem
19   { idx := idx + 1 }
20   this.data := this.data[0..idx] ++ Seq(elem) ++ this.data[idx..]
21 }
```

**Fig. 2.** A sorted list of integers, implemented via immutable sequences. We will discuss implementations based on linked lists and arrays later.

Fig. 2 shows the specification and implementation of a sorted integer list. In this initial version, the list is represented using a mathematical sequence datatype. Line 1 declares an appropriate field; `Int` and `Seq` are built-in datatypes (along with booleans, references, sets and multisets). To make the example more concise, line 3 introduces a parameterised macro that expresses that the argument sequence is sorted.

Viper controls access to the program heap using permissions. Permissions simplify framing (that is, proving that an assertion is not affected by a heap modification), as well as reasoning about concurrency. Permission to a heap location may be held by a method execution or a loop iteration. A method or loop body may access the location only if the appropriate permission is held at the corresponding program point.

Permissions may be transferred between method executions and loop iterations; the permissions to be transferred are specified as part of method pre- and postconditions, and loop invariants, respectively. These specifications are based on implicit dynamic frames [36, 24, 31]. The most fundamental construct is the *accessibility predicate*, `acc(`$e$`.`$f$`)`, which represents permission to a single field location: the field $f$ of the reference denoted by $e$.

Method `insert` in Fig. 2 adds a new element to the list. It returns the index at which the element was inserted, which is useful both programmatically (to retrieve the element later), and to simplify the specified postcondition. The precondition of `insert` requires that callers provide permission to access the list's `data` field; moreover, the list must be sorted. The first postcondition returns the permission to the caller and guarantees that the list remains sorted. The second postcondition constrains the range of the returned index, while the third postcondition specifies the functional behaviour. This postcondition uses an `old` expression to refer to the content of the list in the method pre-state. The specification of `insert` reveals implementation details by referring directly to the `data` field. We will discuss language features that support information hiding and data abstraction in Sec. 4.

The implementation of `insert` iterates over the sequence to determine where to insert the new element. Besides the expected properties, the loop invariant requires a *fractional permission* [7] to `this.data`, denoted by `acc(this.data, 1/2)`. Using a half permission here serves two purposes: first, it allows the loop body to *read* `this.data`; second, leaving the other half permission in the method execution enclosing the loop lets the verifier conclude that the loop does not modify `this.data` (for which the full permission is necessary); that is, it can frame properties of this location such as sortedness of the sequence across the loop.

Viper supports a flexible permission model which includes fractional permissions, symbolic permissions via permission-typed variables (of the built-in type `Perm`), and an approach to constrain such symbolic permissions without using concrete fractions [16], which can be used to model counting permissions [8].

## 2.2 Encoding High-level Concepts

The example in the previous subsection shows that Viper can be used to manually specify and verify programs. However, the focus of the language design has mostly been on making Viper an effective intermediate language which can be targeted by a variety of front-ends. To illustrate this use of the language, this subsection presents an encoding of a small client of a sorted list, implemented in a Java-like language.

```
class Client {
  @GuardedBy("this") List l;
  @GuardedBy("this") boolean changed;

  monitor invariant forall int i, j :: 0 <= i && i < j &&
                     j < |l.data| ==> l.data[i] <= l.data[j]
  monitor invariant old(l.data) == l.data || changed

  synchronized void test(int e1, int e2) {
    l.insert(e1);
    l.insert(e2);
    assert l.data[0] <= l.data[1];
    changed = true;
  }
}
```

**Fig. 3.** An example in a Java-like language whose Viper encoding is shown in Fig. 4. We assume here that class `List` has a field `data` whose type is a mathematical sequence. The `@GuardedBy("this")` annotation indicates that the receiver must be locked before accessing the decorated field. The first monitor invariant requires the list to be sorted; the second is a two-state invariant and requires the `changed` flag to be set whenever a thread changes the content of list `l` between acquiring and releasing the monitor.

Class `Client` in Fig. 3 stores a reference to a list in field `l`. We assume here that class `List` has a field `data` whose type is a mathematical sequence; we will show an alternative encoding using mutable arrays in Sec. 3.3. The client is thread-safe and uses coarse-grained locking to protect its data representation (Java's `@GuardedBy("this")` annotation indicates that the receiver must be locked before accessing the field). It maintains two monitor invariants: the first is a one-state invariant that requires the list to be sorted; the second is a two-state invariant which states that any thread that acquires the monitor must either leave the content of the underlying list unchanged or set the `changed` flag to true by the time it releases the monitor. In the latter invariant, we use an `old` expression to refer to the state in which the monitor was acquired. Method `test` acquires the monitor of its receiver (since it is declared `synchronized`), adds two elements to the list and asserts that the first two list elements are in order. It then sets the `changed` flag and implicitly releases the monitor when it terminates.

Guarded command languages such as Boogie encode high-level language features mostly via three primitives: `assert` statements to introduce proof obligations, `assume` statements to state properties which the verifier may use because they have been justified elsewhere, and `havoc` statements to assign non-deterministic values to variables in order to model side effects or interference. Viper provides permission-aware analogues of these primitives: the operation `exhale` $A$ asserts all *pure* assertions in $A$ (that is, assertions that do not include accessibility predicates). Any permissions specified in $A$ via accessibility predicates are *removed* from the current program state; if no permission is left for a location then no information about its value is retained, similarly to havocking the location. Conversely, `inhale` $A$ assumes all pure assertions in $A$ and *adds* permissions.

```
 1  field changed: Bool
 2  field l: Ref
 3  field held: Int
 4
 5  method test(this: Ref, e1: Int, e2: Int)
 6    ensures [true, forperm[held] r :: false]
 7  {
 8    // acquire l
 9    inhale acc(this.l) && acc(this.l.data) && acc(this.changed) &&
10           sorted(this.l.data)
11    inhale acc(this.held)
12  statelabel acq
13
14    var tmp: Int
15    tmp := insert(this.l, e1)
16    tmp := insert(this.l, e2)
17    assert this.l.data[0] <= this.l.data[1]
18    this.changed := true
19
20    // release l
21    exhale acc(this.l) && acc(this.l.data) && acc(this.changed) &&
22           sorted(this.l.data) &&
23           (old[acq](this.l.data) == this.l.data || this.changed)
24    exhale acc(this.held)
25  }
```

**Fig. 4.** A simplified Viper encoding of the source program in Fig. 3.

Fig. 4 shows a simplified Viper encoding of the client from Fig. 3, using `exhale` and `inhale` to encode concurrency features, which are not supported by Viper directly. We model locks as resources which can be transferred between methods. To model this, the Viper program includes a field `held` and uses the permission to location $o$`.held` to represent that the monitor of object $o$ is held by the current method execution. Consequently, acquiring the receiver's monitor at the start of method `test` is encoded by inhaling permission to `this.held` (line 11), and releasing the monitor exhales this permission (line 24). This encoding ensures that a monitor can be released only when it is held. We do not include checks for other properties such as deadlock freedom here, but they could also be encoded.

Note that the only purpose of field `held` is to use its permission to represent that a monitor is held; its value and type are irrelevant.

We encode the `@GuardedBy` annotations by inhaling permission to the client's fields when acquiring the monitor (line 9) and exhaling them upon release (line 21). We interpret `@GuardedBy` deeply and include the permission to the list's `data` field. Finally, the encoding of acquire and release also takes into account the monitor invariants declared in the source program. Acquiring a monitor inhales its (one-state) invariant (line 10). Releasing it exhales the one-state and two-state invariants (lines 22–23). Checking a two-state invariant requires a way to access the earlier of the two states: here, the state in which the monitor was acquired. Viper provides a convenient way to refer to earlier program states: programs can declare *state labels* (line 12) and refer to these states in later assertions using labelled `old` expressions (line 23). This feature is also useful for encoding other comparisons across states such as termination measures.

It is often useful to assert or assume properties about the permissions currently held, without adding or removing permission. Viper supports this via two pure assertions: `perm(`$o.f$`)` yields the permission amount held for location $o.f$ in the current state; `forperm[`$f$`]` $r$ `::` $P(r)$ expresses that all references $r$ to which the current state has non-zero permission to $r.f$, satisfy $P(r)$. The example in Fig. 4 uses the latter feature to encode a *leak check* for monitors; this check fails if a method terminates without either releasing the monitors that it holds or explicitly transferring them back to the caller via a postcondition. The leak check is expressed by the assertion `forperm[held] r :: false` in line 6.

Since the leak check must be performed *after* any remaining monitors have been transferred to the caller via the method's postcondition, it cannot be placed at the end of the method body, where it would be checked *before* exhaling the postcondition. Therefore, we place it in a postcondition and encode it as *inhale-exhale assertion.* These assertions have the form `[`$A_1$`,`$A_2$`]` and are interpreted as $A_1$ when the assertion is inhaled and $A_2$ when the assertion is exhaled. In our example, the leak check is performed during exhale, but no corresponding assumption is made by the caller when inhaling the postcondition after a call.

It is common for encodings of high-level verification techniques to contain asymmetries between the properties that are assumed and those that are checked. The leak check is an example of a property that is checked, but not assumed. It is also common to assume properties that are justified by a different (possibly weaker or even vacuous) check together with an external argument provided by a type system, soundness proof or other meta-reasoning. For instance, the following assertion allows the verifier to use a quantified property in its direct form when assuming the property, and to use the premises of the corresponding inductive argument when proving the property:

```
[forall x: Int :: 0 <= x ==> P(x) ,
 forall x: Int :: (forall y: Int :: 0 <= y && y < x ==> P(y)) &&
                                                   0 <= x ==> P(x)]
```

## 3 Unbounded Heap Structures

Viper supports several idioms for specifying and reasoning about unbounded heap structures. There are no specific definitions built in; instead, Viper includes three features which allow one to provide the relevant definitions as part of the input program: recursive predicates (the traditional means in separation-logic-based tools), magic wands (useful for specifying data structures with "missing parts"), and quantified permissions (for writing pointwise rather than recursive specifications). We will briefly discuss each of these features in this section, with respect to variations on our example in Fig. 2. We will focus on the specification of permissions, and show how to extend these specifications with sortedness constraints and rich functional properties in Sec. 4 and the online appendix [28].

### 3.1 Recursive Predicates

*Recursive predicates* [30] are the classical means in separation logic of specifying linked data structures such as lists and trees. A predicate definition consists of a name, a list of formal parameters, and a body, which contains the assertion defining the predicate. The body is optional; omitting it results in an abstract predicate, which is useful to hide implementation details from client code. Like permissions, predicates may be held by method executions and loop iterations, and may be transferred between them. Exchanging a predicate for its body and vice versa is done via `unfold` and `fold` statements to prevent the automatic prover from unfolding a recursive definition indefinitely. In expressions, `unfolding` can be used to temporarily unfold a predicate.

```
field data: Ref // for the nodes
field next: Ref // for the nodes
field head: Ref // for the list head

predicate List(this: Ref)
{
  acc(this.head) && acc(lseg(this.head, null))
}

predicate lseg(this: Ref, end: Ref)
{
  this != end ==>
    acc(this.data) && acc(this.next) && acc(lseg(this.next, end))
}
```

**Fig. 5.** Fields and predicates for a linked list structure. The `acc` syntax around predicate instances is optional, but needed when specifying fractional permissions to predicates.

As an example, we consider a variant of Fig. 2, in which the list is implemented based on a linked list of nodes. The appropriate predicate definitions can be found in Fig. 5. The `List` predicate provides the definition for the permissions to an entire instance of the list. It is defined in terms of the `lseg` predicate, which defines a list *segment* from `start` to `end`: in this case, from `this.head` to `null`.

List segment predicates can be used to specify iterative traversals of linked lists, as shown in Fig. 6. The loop invariant at lines 20-21 describes the permissions to the list nodes in terms of one `lseg` predicate for the nodes seen so far and one for the remainder of the list. The former explains the need for a list segment predicate; tracking permissions for the partial list already inspected is needed to reassemble the whole list after the loop (the code to do this is omitted at line 29).

Manipulating recursive predicates can be tedious. While it is easy to prepend an element to a data structure (by folding another instance of the predicate), extending a data structure at the other end requires additional work to unfold the recursive predicate until the end and then re-fold it including the new element. In Fig. 6, this operation is performed by the `concat` method, which plays the role of proving the lemma that from `lseg(x,y) && lseg(y,z)` we can obtain `lseg(x,z)`. `concat` is a specification-only method, but Viper does not distinguish between regular and ghost code. In the next subsection, we will explain an approach that reduces the overhead of writing and proving such methods in many cases.

## 3.2 Magic Wands

The *magic wand* is a binary connective (written $A \mathrel{-\!\!*} B$), which describes the promise that if combined with state satisfying the assertion $A$, the combination can be exchanged for the assertion $B$ [29, 34].

Fig. 7 shows an alternative specification of the loop from Fig. 6 (lines 17-31). The alternative loop invariant uses a magic wand to represent the permissions to the partial list seen so far. These permissions are expressed indirectly, by the promise that the wand can be combined with the permission to the remainder of the list (the list segment `acc(lseg(ptr,null))`) to obtain permission to the full list. The permissions implicitly associated with the magic wand instance are essentially the same as those required by the `acc(lseg(hd,ptr))` assertion in Fig. 6, which is replaced by the wand.

Viper's support for magic wands [35] includes heuristics to automate (in many cases) reasoning about magic wand assertions, for example, in establishing our loop invariant. Magic wands can also be manipulated manually via dedicated operations, similar to the `fold` and `unfold` statements used for predicates [35]. For example, the `apply` statement in line 11 of Fig. 7 instructs the verifier to exchange the magic wand assertion and its left-hand side for the right-hand-side, restoring the full list after the (partial) traversal.

Compared to the solution without magic wands in Fig. 6, we no longer require the auxiliary `concat` method to manage `lseg` predicates. In addition, we could replace `lseg` by a simpler predicate that describes only full lists. Magic wands provide a general means for tracking partial versions of data structures, without the need to explicitly define or manipulate these partial versions.

```
method insert(this: Ref, elem: Int) returns (idx: Int)
  requires acc(List(this))
  ensures  acc(List(this))
{
  idx := 0;  var tmp: Ref
  unfold acc(List(this))
  if(this.head != null) { unfold acc(lseg(this.head, null)) }

  if(this.head == null || elem <= this.head.data)
  {
    ... // allocate new node at this.head, fold predicates
  } else {
    var hd : Ref := this.head
    var ptr: Ref := hd // running variable
    idx := idx + 1

    fold acc(lseg(hd, hd))  // for loop invariant
    while(ptr.next != null &&
        unfolding acc(lseg(ptr.next, null)) in ptr.next.data < elem)
      invariant acc(lseg(hd, ptr)) && acc(ptr.next) && acc(ptr.data)
      invariant acc(lseg(ptr.next, null))
    {
      unfold acc(lseg(ptr.next, null))
      idx := idx + 1;  var ptrn: Ref := ptr.next
      fold acc(lseg(ptrn, ptrn));  fold acc(lseg(ptr, ptrn))
      concat(hd, ptr, ptrn) // add to end of list segment
      ptr := ptrn
    }
    ... // allocate new node at ptr.next, fold predicates
    concat(hd, ptr, null) // concat two lsegs to obtain full list
  }
  fold acc(List(this))
}

method concat(this: Ref, ptr: Ref, end: Ref)
  requires acc(lseg(this, ptr)) && acc(lseg(ptr, end))
  requires end != null ==> acc(end.next, 1/2) // not forming a cycle
  ensures  acc(lseg(this, end))
  ensures  end != null ==> acc(end.next, 1/2)
{
  if(this != ptr) {
    unfold acc(lseg(this, ptr));  concat(this.next, ptr, end)
    fold acc(lseg(this, end))
  }
}
```

**Fig. 6.** The insert method of a sorted linked list with recursive predicates.

```
      unfold acc(lseg(ptr.next, null))
      idx := idx + 1;  var last: Ref := ptr
      ptr := ptr.next
    }
     // allocate new node at ptr.next, fold predicates
    tmp := new(data,next)
    tmp.data := elem
    tmp.next := ptr.next
    ptr.next := tmp
    fold acc(lseg(ptr.next, null));  fold acc(lseg(ptr, null))
    apply acc(lseg(ptr, null)) --* acc(lseg(hd, null)) // full list
```

**Fig. 7.** Alternative loop specification with magic wands (cf. Fig. 6, lines 17-31).

## 3.3 Quantified Permissions

In addition to recursive predicates, Viper supports *quantified permissions* as a means of specifying unbounded heap structures. Quantified permissions are similar to separation logic's iterated separating conjunction [34] and allow the specification of permissions *pointwise*. The flat structure of a pointwise specification is convenient for specifying data structures that are not limited to traversals in a single, hierarchical fashion, such as cyclic lists, random access data structures such as arrays, and general graphs.

We denote quantified permissions by a universal quantifier around the usual accessibility predicates. For example, `forall x: Ref :: x in S ==> acc(x.f)` denotes permission to the `f` field of every reference in the set `S`. The quantified variable can be of any type, and we permit arbitrary boolean expressions to constrain its range.

Quantified permissions provide a natural way to specify properties of arrays. Arrays are not supported natively in Viper but can be encoded. As we show in Sec. 5, we can introduce a custom type `Array` which models the $i$th slot of an array $a$ as `loc(`$a$`,`$i$`).val`, where `loc(a: Array, i: Int): Ref` is an injective function provided by the `Array` type. The type also provides a function `len(a: Array): Int` to model the length of an array. One can then denote permission to the array slots via quantified permissions ranging over the array indices.

Fig. 8 applies this approach to encode an array list. The field `elems` stores the array, while `size` keeps track of the number of used array slots. The quantified permission assertion at line 9 represents permission to all array slots. These are used, for instance, to permit the array access in the while-condition in line 20. Note that the loop invariant is essentially a copy of the `AList` predicate body (with the additional constraint on the `idx` loop variable). We employ fractional permissions (including fractional quantified permissions in line 23) to specify that the loop will not modify the corresponding locations.

```
field val: Int     // array slots modelled by loc(this.elems,i).val
field elems: Array // see Array domain definition in Sec. 5
field size: Int    // how many array slots have been used

predicate AList(this: Ref)
{
  acc(this.elems) && acc(this.size) &&
  0 <= this.size && this.size <= len(this.elems) &&
  (forall i: Int :: 0 <= i && i < len(this.elems) ==>
                                    acc(loc(this.elems, i).val))
}

method insert(this: Ref, elem: Int) returns (idx: Int)
  requires acc(AList(this))
  ensures  acc(AList(this))
{
  idx := 0
  unfold acc(AList(this))

  while (idx < this.size && loc(this.elems, idx).val < elem)
    invariant acc(this.elems, 1/2) && acc(this.size, 1/2)
    invariant this.size <= len(this.elems)
    invariant forall i: Int :: 0 <= i && i < len(this.elems) ==>
                             acc(loc(this.elems, i).val, 1/2)
    invariant 0 <= idx && idx <= this.size
  { idx := idx + 1 }

  ... // move the later elements forward by one, resize if necessary
  loc(this.elems, idx).val := elem
  this.size := this.size + 1
  fold acc(AList(this))
}
```

**Fig. 8.** Array-list, specified using quantified permissions.

## 4 Functional Behaviour

The specifications shown in Sec. 3 focus on the management of permissions, but do not constrain the values stored in data structures (for instance, to require sortedness of the list) or computed by operations (for instance, to express the functional behaviour of method `insert`). The examples in Sec. 2 specify such properties, but in a way which exposes implementation details. In this section, we explain several ways to express functional behaviour in Viper.

A simple way to specify the values stored in data structures is to include constraints on the values in the body of a predicate, in addition to permissions. For example, we could extend the body of the `lseg` predicate in Fig. 5 by conjoining the following assertion:

```
  unfolding acc(lseg(this.next, end)) in
    this.next != end ==> this.data <= this.next.data
```

This assertion specifies sortedness pairwise between list nodes. Maintaining the augmented predicate entails corresponding additions to the loop invariant and specification of the `concat` method in Fig. 6, as shown in the online appendix.

Constraining values via predicates allows one to encode representation invariants, but is not suitable to express client-visible invariants or the functional behaviour of operations. To support such specifications, Viper supports *heap-dependent functions* that may be used in program statements and assertions. Functions (as opposed to methods) have (side-effect free) expressions rather than statements as a body. A function's precondition must require sufficient permissions to evaluate the function's body; in contrast to methods, invoking a function does not consume these permissions, and they do not need to be returned via a function's postcondition.

Functions are a flexible feature which can play several different roles in a Viper program. The first major role is to encode side-effect free observer methods (*pure* methods in JML [21] and Spec# [1]), which are a part of the interface of many data structures. For instance, list-style collections typically provide observer methods such as `length` and `itemAt` to retrieve data. As an example, we extend our `lseg`-based specification from Sec. 3.1 with the following function definition:

```
function lengthNodes(this: Ref, end: Ref): Int
  requires acc(lseg(this, end))
{
  unfolding acc(lseg(this, end)) in
    this == end ? 0 : 1 + lengthNodes(this.next, end)
}
```

This definition enables us, whenever we hold an `lseg` predicate instance, to express its length via an application of `lengthNodes`. The Viper verifiers carefully (and automatically) control the unrolling of recursive function definitions, essentially mimicking the traversal of the corresponding `lseg` data structure [15].

A second major role of functions is to define *abstraction functions* [17] providing abstractions of the underlying data representation, in order to express specifications without revealing implementation details. For example, the following function abstracts the values of a list segment to a mathematical sequence:

```
function contentNodes(this: Ref, end: Ref): Seq[Int]
  requires acc(lseg(this, end))
  ensures  forall i: Int, j: Int :: 0 <= i && i < j && j < |result|
                               ==> result[i] <= result[j]
{
  this == end ? Seq[Ref]() : unfolding acc(lseg(this, end)) in
                ( Seq(this.data) ++ contentNodes(this.next, end) )
}
```

Viper verifiers reason about function applications in terms of the function's body. Nevertheless, it is sometimes useful to provide a function postcondition. In the above example, the postcondition expresses that the sequence of all values stored in the list is sorted, which is implied by the pairwise sortedness we have added to the `lseg` predicate. Note that the inductive argument required to justify this postcondition is implicit in the checking of `contentNodes`'s recursive definition.

A similar `content` function for the overall data structure (described by the `List` predicate) allows us to specify the functional behaviour of `insert`:

```
  ensures content(this) == old(content(this))[0..index] ++
                           Seq(elem) ++ old(content(this))[index..]
```

Function bodies are optional in Viper, which allows hiding details when verifying client code (similarly to predicates). Omitting the body is also useful for axiomatising a function rather than defining it (assuming the existence of the function is otherwise justified). In the array list example from Fig. 8, defining `length` and `itemAt` functions is straightforward. However, an analogous `content` function would be awkward to define recursively since our specifications for this random-access example avoid recursive definitions. Instead, we can axiomatise the function, that is, specify its meaning via a quantified postcondition. Such quantifiers are supported in Viper assertions in general, and provide another important tool for writing functional specifications:

```
function content(this: Ref): Seq[Int]
  requires acc(AList(this))
  ensures  |result| == length(this)
  ensures  forall i: Int :: 0 <= i && i < length(this)
                     ==> result[i] == itemAt(this, i)
```

The third major role of heap-dependent functions is to express refinements of existing predicate definitions. For example, instead of expressing sortedness as part of a predicate definition, we can write a boolean function (here for the array list from Fig. 8) and use it in combination with the unchanged `AList` predicate:

```
function sorted(this: Ref): Bool
  requires acc(AList(this))
{
  unfolding acc(AList(this)) in
      forall i: Int, j: Int :: 0 <= i && i < j && j < this.size
                        ==> result[i] <= result[j]
}
```

`AList(this) && sorted(this)` describes a sorted list, while `AList(this)` specifies an array list that may or may not be sorted. In this way, functions can be used to augment data-structure instances with additional invariants, without requiring many versions of a predicate definition or resorting to higher-order logic.

The combination of predicates, functions, and quantifiers supported by Viper provides the means for writing rich functional specifications in a variety of styles, which are further illustrated by examples in the online appendix [28].

## 5 First-Order Theories

Many specification and verification techniques provide their own mathematical vocabulary, for instance, to encode algebraic data types. To support such techniques, Viper supports the declaration of custom first-order theories via *domains*: each domain introduces a (potentially polymorphic) type and may declare uninterpreted function symbols and axioms. Organising mathematical theories into domains allows back-ends to provide dedicated support for certain theories. For instance, while both Viper verifiers let the underlying SMT solver reason about domains, an abstract-interpretation-based inference might provide specialised abstract domains for certain Viper domains.

```
domain Array {
  function loc(a: Array, i: Int): Ref
  function len(a: Array): Int
  function loc_a(r: Ref): Array
  function loc_i(r: Ref): Int

  axiom loc_injective {
    forall a: Array, i: Int :: {loc(a, i)} 0 <= i && i < len(a)
       ==> loc_a(loc(a, i))) == a && loc_i(loc(a, i))) == i
  }

  axiom length_nonneg { forall a: Array :: 0 <= len(a) }
}
```

**Fig. 9.** A domain definition for arrays, as used in Sec. 3.3. The injective function `loc` maps an array and an index to a reference; in combination with a field (such as `val` in Fig. 8), an array slot `a[i]` can be encoded as `loc(a, i).val`.

Fig. 9 uses a domain to model arrays, which are not natively supported in Viper. We represent the $i$th slot of an array $a$ as `loc(`$a$`,`$i$`).val`, where `loc` is a function introduced by the domain and `val` is a suitable field. Since each array slot corresponds to a dedicated memory location, `loc` must be injective; this property is expressed by the axiom `loc_injective`, which axiomatises `loc_a` and `loc_i` as the inverse functions of `loc`. Axiomatising injectivity via inverse functions improves performance of the SMT solver by reducing the number of instantiations of the axiom.

Universal quantifiers in axioms (as well as in assertions) may be decorated with triggers [27]: terms used as patterns which restrict the potential instantiations. For instance, the trigger `{loc(a, i)}` in axiom `loc_injective` lets the SMT solver instantiate the quantifier with $x$ and $y$ whenever it knows about a term `loc(`$x$`,`$y$`)`. When no trigger is provided, Viper attempts to infer triggers automatically. In general, however, hand-crafted triggers lead to better performance.

The online appendix [28] shows how to encode algebraic data types as domains, with functions for constructors and selectors, and with appropriate axioms. Such an encoding is useful when encoding source languages that provide ADTs (such as Scala's case classes) or for specification languages that make use of ADTs.

## 6 Evaluation

In this section, we evaluate the performance of the Viper verifiers on a wide variety of examples. Moreover, we give preliminary qualitative and quantitative evidence for Viper's suitability as an intermediate verification language.

### 6.1 Performance of the Viper Verifiers

To evaluate the performance of the Viper verifiers, we ran both our symbolic execution (SE) verifier and our verification-condition-generation (VCG) verifier on the following collections of input programs: our own Viper regression tests, Viper programs generated by the VerCors tools [5, 4], and programs generated from Chalice examples via our Chalice front-end. For the Viper and VerCors programs, we split the files into those using quantified permissions (for which only our SE verifier currently provides support), and those which can be run in both verifiers. The set of VerCors examples was provided to us by the VerCors developers as representative of their Viper usage.

The results are shown in Fig. 10. Both verifiers perform consistently well in the average case, with the SE verifier being significantly faster. As the average times suggest, the maximum times are true outliers—these were typically examples designed to be complex, in order to test what the tools could handle. The Viper tests (which are mostly regression tests) tend to be shorter and less challenging than the VerCors-generated programs, which are representative of real usage of Viper as a back-end infrastructure.

|  | Number of | Average | Mean time (s) |  | Max. time (s) |  |
|---|---|---|---|---|---|---|
| Input programs | programs | size (LOC) | SE | VCG | SE | VCG |
| Viper tests w/o QPs | 208 | 43.8 | 0.23 | 0.81 | 18.36 | 34.17 |
| VerCors w/o QPs | 43 | 152.1 | 0.94 | 2.24 | 16.25 | 31.78 |
| Chalice (no QPs) | 221 | 122.0 | 0.26 | 0.97 | 21.26 | 29.37 |
| Viper tests with QPs | 74 | 34.0 | 0.30 | - | 2.00 | - |
| VerCors with QPs | 65 | 105.6 | 0.95 | - | 8.39 | - |

**Fig. 10.** Performance evaluation of Viper verifiers. Lines of code (LOC) measurements do not include whitespace lines and comments. All input programs were run 10 times and average times recorded. The mean and maximum times were calculated based on these averages. Timings do not include JVM start-up time: we persist a JVM across test runs using the Nailgun tool; for the VCG verifier, timings include start-up of Boogie via Mono. All timings were gathered on a Lenovo Thinkpad T450s running Ubuntu 15.04 64 bit, with 12GB RAM; full details are available in our online appendix [28].

### 6.2 Viper as an Intermediate Verification Language

To assess Viper's suitability as an intermediate verification language, we provide some observations about Viper's language design and compare the performance of Viper as the back-end of the VerCors tools. to the previously-used Chalice-Boogie tool chain [26].

**Language Design.** The most comprehensive front-ends for Viper are the Java and OpenCL front-ends developed in the VerCors project and our own Chalice/Viper front-end. Various language features of Viper have proven essential for these different front-ends. VerCors' work on verifying concurrent Java makes use of Viper's custom domains for encoding custom ADT-like datatypes along with additional axioms, and makes heavy use of sequences, recursive functions and predicates. The VerCors OpenCL front-end instead employs quantified permissions along with domains similar to the array encoding shown in Sec. 5, and pure quantifiers to specify functional properties. Our front-end for Chalice makes extensive use of `inhale` and `exhale` statements to encode high-level features, similarly to the example in Sec. 2.2. As such, the key language features described in this paper have all been heavily used in at least one existing front-end.

There are Chalice front-ends for both Boogie and Viper, which support very similar (but not identical) versions of the Chalice language. For the Chalice programs from the previous subsection, the Boogie files were between 3.3 and 32.1 times the size of the corresponding Viper files, and on average 11.2 times larger. This significant difference illustrates the higher level of abstraction provided by the Viper language, compared with existing intermediate verification languages.

**Performance of the Infrastructure.** The VerCors project switched from using Chalice-Boogie as back-end infrastructure, to Viper. This switch was partly motivated by the available language features; for instance, the VerCors OpenCL front-end relies heavily on quantified permissions, which are not available in Chalice. Another reason was the performance of the Viper tools. In the following, we compare the performance of the two infrastructures on inputs generated by the VerCors tools.

Running tests through the entire alternative tool chains proved difficult due to legacy syntactic and implementation differences; however, we identified 17 VerCors examples from the test suite used in Sec. 6.1 that could be run on the alternative infrastructures. For each of these examples, we generated two (essentially equivalent) Boogie programs, one using Chalice as a VerCors back-end, and one using Viper with our VCG verifier.

Fig. 11 shows the results of our comparison. In all cases, the Boogie files generated via the Viper route were smaller and verified faster. The same example was slowest via both routes, and more than 4 times faster in the Viper-generated version. Although our sample size is small, the results suggest Viper enables a more direct encoding and offers a more streamlined verification condition generator. In practice, the VerCors team typically use Viper's SE verifier, which is substantially faster still, as shown in Fig. 10.

|  | Average size (LOC) | Mean time (s) | Max. time (s) |
|---|---|---|---|
| Boogie file via Chalice | 945.0 | 0.83 | 3.22 |
| Boogie file via Viper (VCG) | 631.1 | 0.53 | 0.73 |
| Ratio | 66.8% | 64.3% | 22.5% |

**Fig. 11.** Comparison of alternative back-end infrastructures for the VerCors tools. Using Viper's VCG verifier significantly reduces the size and verification time of the generated Boogie programs compared to the Chalice/Boogie infrastructure.

## 7 Related Work

Boogie [22] and Why [14] are widely-used intermediate verification languages, but they do not offer native support for permission-based reasoning. Chalice [26] demonstrates that permissions can be encoded in such a first-order setting; our VCG-based back-end makes such a complex encoding reusable. Boogie and Why front-ends encode heaps as maps. In contrast, the Viper language has a built-in notion of heap, which is slightly less expressive (for instance, in Viper, heaps cannot be stored in variables), but enables the development of more-specialised back-ends, such as verifiers based on Smallfoot-style symbolic execution and inference engines based on abstract interpretation.

To our knowledge, the only other verification infrastructure for permission-based reasoning is coreStar [6], which includes an intermediate language for separation logic and a symbolic execution engine. Front-ends implemented on top of coreStar encode programs into coreStar's language and also need to provide proof rules and abstraction rules to customise the behaviour of coreStar's symbolic execution, even for fundamental concepts such as permissions (points-to predicates). In contrast, Viper has been designed to be expressive enough to capture a wide variety of languages and verification techniques out of the box, without requiring front-end developers to descend into the back-end(s). Furthermore, having a fixed language (with fixed rules) simplifies writing different back-ends, potentially with specialised handling of certain language features.

Some verifiers for separation logic such as Smallfoot [3], GRASShopper [33], Asterix [32], and the work by Chin et al. [9], achieve a relatively high degree of automation by restricting themselves to specific (classes of) theories: often those of linked lists and trees. Without support for important features such as fractional permissions or user-defined predicates and functions, they do not offer the expressiveness needed for an intermediate language which can encode a wide range of verification techniques.

VeriFast [18], a verifier for C and Java programs, supports an expressive assertion language, including user-defined higher-order predicates and function pointers, but it requires significant amounts of user annotations, in particular when reasoning about functional specifications and abstractions. This complicates the encoding of front-end languages that try to achieve a higher degree of automation.

Several verification techniques based on interactive proof assistants such as Coq or HOL4 [2, 10, 12, 37] provide tactics that automate common proof steps in separation logic. Viper aims at a higher level of automation, such that users do not have to interact directly with the verification back-ends.

## 8 Conclusion and Future Work

We have presented Viper, an infrastructure which facilitates the rapid prototyping of permission-based verification techniques and the development of verification tools. Viper's intermediate language offers a flexible permission model, supports user-defined predicates and functions, and provides advanced specification features such as magic wands and quantified permissions. It provides the necessary expressiveness to encode a wide range of language features and permission-based verification techniques. In particular, users may choose between and combine different styles of encodings, as we have demonstrated in Sec. 3 and Sec. 4. Viper includes two back-end verifiers: one based on verification condition generation and one based on symbolic execution. An abstract-interpretation-based specification inference is under development.

Viper is targeted by several front-ends, developed both inside and outside of our research group. Together with collaborators, we are currently working on encodings of verification techniques for JavaScript and for fine-grained concurrency. Viper is also being used to verify safety and security properties of a network router implemented in Python.

As future work, we plan to provide a comprehensive variety of specification inference techniques and to improve the reporting and debugging of verification failures. We are also interested in integrating alternative, possibly specialised verification back-ends.

**Acknowledgements.** This work has been funded in part by the Swiss National Science Foundation and by the Hasler Foundation. Viper has benefitted from the work of many people. We are especially grateful to Uri Juhasz for his contributions to the design of the Viper language, Ioannis Kassios for his work on quantified permissions and the initial Chalice front-end, and to Vytautas Astrauskas for his improvements of the latter. We are also thankful to Milos Novacek for his work on specification inference, and to Stefan Heule for the initial development of the verification condition generator. We warmly thank Stefan Blom for his extensive feedback on Viper and his help with using VerCors, and Marco Eilers for his help with our experimental evaluation.

## 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. Bengtson, J. B. Jensen, and L. Birkedal. Charge! - A framework for higher-order separation logic in Coq. In L. Beringer and A. P. Felty, editors, *ITP*, volume 7406 of *LNCS*, pages 315–331. Springer, 2012.
3. 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.
4. S. Blom, S. Darabi, and M. Huisman. Verification of loop parallelisations. In A. Egyed and I. Schaefer, editors, *FASE*, volume 9033 of *LNCS*, pages 202–217. Springer, 2015.
5. S. Blom and M. Huisman. The VerCors tool for verification of concurrent programs. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *FM*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.
6. 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 2011*, pages 65–77, 2011. Available at http://research.microsoft.com/en-us/um/people/moskal/boogie2011/boogie_2011_all.pdf.
7. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *SAS*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
8. J. T. Boyland, P. Müller, M. Schwerhoff, and A. J. Summers. Constraint semantics for abstract read permissions. In *FTfJP*, FTfJP'14, pages 2:1–2:6, New York, NY, USA, 2014. ACM.
9. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, Aug. 2012.
10. A. Chlipala, J. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In G. Hutton and A. P. Tolmach, editors, *ICFP*, pages 79–90. ACM, 2009.
11. 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.
12. R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In Z. Hu, editor, *APLAS*, volume 5904 of *LNCS*, pages 161–177. Springer, 2009.
13. 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.
14. 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.
15. 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.
16. S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers. Abstract read permissions: Fractional permissions without the fractions. In *VMCAI*, LNCS. Springer, 2013.
17. C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.
18. 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, *NFM*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011.

19. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.
20. 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.
21. 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.
22. K. R. M. Leino. This is Boogie 2. Working draft; available at http://research.microsoft.com/en-us/um/people/leino/papers.html, 2008.
23. 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.
24. 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.
25. K. R. M. Leino and P. Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In P. Müller, editor, *LASER Summer School 2007/2008*, volume 6029 of *LNCS*, pages 91–139. Springer, 2010.
26. 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.
27. M. Moskal. Programming with triggers. In *SMT*, SMT '09, pages 20–29, New York, NY, USA, 2009. ACM.
28. P. Müller, M. Schwerhoff, and A. J. Summers. Online appendix. Available from http://viper.ethz.ch/VMCAI16.
29. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.
30. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.
31. M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3:01):1–54, 2012.
32. 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.
33. 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.
34. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
35. M. Schwerhoff and A. J. Summers. Lightweight support for magic wands in an automatic verifier. In J. T. Boyland, editor, *ECOOP*, volume 37 of *LIPIcs*, pages 614–638. Schloss Dagstuhl, 2015.
36. 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.
37. T. Tuerk. A formalisation of Smallfoot in HOL. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, pages 469–484. Springer, 2009.