Master Thesis Report

From Viper to GRASShopper

Translating Between Intermediate Languages for Software Verification

Andrea Helfenstein

Supervisors: Dr. Alexander J. Summers, Prof. Dr. Peter Müller

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

2016-09-07

Contents

1 Introduction

The Viper project [1], developed at ETH Zurich, provides a number of tools for verifying programs written in the intermediate verification language Viper. Verification problems of higher level concurrent programming languages can be encoded in Viper. Currently there are two backend verifiers available to check Viper programs. One backend is based on verification condition generation, while the other is based on symbolic execution.

The GRASShopper tool [2], developed at New York University (NYU), can verify programs at a similar level of abstraction. It is based on their own GRASS logic [3, 4], which is a logic based on reachability in graphs.

In this report we demonstrate how a subset of the Viper language can be translated to the GRASShopper input language[1]. This is particularly interesting because although both tools are based on the ideas of separation logic and can verify programs operating on a heap, their approaches for how to describe recursive data structures, such as linked lists, are fundamentally different. In Viper, such data structures are traditionally described using recursive predicates. Recently, support for point-wise definitions has been added to Viper [5]. This allows data structures such as doubly-linked lists or general graphs to be described more easily. In GRASShopper, recursive data structures are defined using reachability predicates, expressing e.g. that a node in a linked list can be reached from the head of the list.

Our new tool Natrix is an alternative backend verifier for Viper, that uses the translation discussed in this report to verify a subset of the Viper language using the GRASShopper tool.

[1] We refer to this language simply as *GRASShopper* from now on.

2 Background

In this section we describe the language features of both Viper and GRASShopper that are relevant to this report. Although both Viper and GRASShopper are based on separation logic, the languages have several differences that are important when translating one language to the other.

2.1 Viper

Here we explain the features of the Viper language relevant to this report. More detailed information on Viper can be found in [1].

Reference Types Viper has a single reference type `Ref`. The available fields are the same for all objects. Fields are defined as top-level members of the program: a field `f` of type `Int` would be defined as `field f: Int`. Viper manages permissions to each pair of reference and field separately.

Permissions In Viper, a field access $x.f$ only succeeds if we have permission to field f of reference x. Viper distinguishes between read and write permissions. Permission amounts are modeled as fractional amounts ranging from 0 to 1, where 0 represents no permission (also referred to as `none`), 1 represents full permission (`write`), and amount in between is a read permission. In assertions, the accessibility predicate `acc` $(x.f, p)$ is used to check that we have permission amount p for field f of reference x. `acc` $(x.f)$ is used as a shorthand for `acc` $(x.f,$ `write` $)$. Assertions that contain no accessibility predicates are called *pure*.
Viper also supports so-called *quantified permission* assertions, written as

$$\texttt{forall}\ x : T :: b(x) \texttt{ ==> acc}\ (e(x).f, p(x)),$$

where $b(x)$ is a Boolean expression, $e(x)$ is an expression referring to a reference, and $p(x)$ is a permission amount. The assertion means that for each x where $b(x)$ holds we have $p(x)$ amount of permission to field f of $e(x)$.

Checking Permissions In Viper, `assert` statements and contracts check that the permissions they require are a subset of the permissions that are available at this point in the program. Assume we have full permissions to `x.f` and `y.f` at some point in a program. A statement `assert acc(x.f)` will succeed at this position in the program, since we have full permission to `x.f`.

Predicates and Recursive Data Structures In Viper, a predicate is specified as

$$\texttt{predicate}\ p(\mathit{args})\ \texttt{\{}\ e\ \texttt{\}},$$

where *args* is a list of typed parameters, and e a potentially recursive assertion in terms of the parameters. Predicates are used to describe properties of their arguments. They may be held like permissions. We use `unfold` and `fold` statements to exchange the predicate for its body. There is also an `unfolding` expression, that temporarily unfolds a predicate.
Recursive data structures such as linked lists can be specified using recursive predicates in Viper. See listing 1 for an example definition of a linked list segment using a recursive predicate. This example also illustrates the behavior of `unfold` and `fold` statements.

```
field data: Int
field next: Ref

predicate lseg(this: Ref, end: Ref)
{
  this != end ==>
  acc(this.data) && this.data >= 0 && acc(this.next) &&
  acc(lseg(this.next, end))
}

method m1(l: Ref)
  requires acc(lseg(l, null)) && l != null
  ensures acc(lseg(l, null)) && l != null
{
  assert l.data >= 0 // fails
}

method m2(l: Ref)
  requires acc(lseg(l, null)) && l != null
  ensures acc(lseg(l, null)) && l != null
{
  unfold acc(lseg(l, null))
  assert l.data >= 0 // succeeds
  fold acc(lseg(l, null))
}

method m3(l: Ref)
  requires acc(lseg(l, null)) && l != null
  ensures acc(lseg(l, null)) && l != null
{
  assert unfolding acc(lseg(l, null)) in l.data >= 0 // succeeds
}
```

Listing 1: Recursive definition of a linked list in Viper with positive data elements. The methods try to access the first element of the list, illustrating the use of fold and unfold statements.

```
function sum(x: Ref): Int
  requires acc(x.f) && acc(x.g)
  ensures result == x.g + x.f
{
  x.f + x.g
}
```

Listing 2: A Viper function computing the sum of two fields

Functions Viper distinguishes between heap-dependent and heap-independent functions. Functions that appear as top-level members of the program are potentially heap-dependent. Those functions have assertions as preconditions and as the body. The postconditions must be pure expressions, i.e. they cannot contain accessibility predicates. See listing 2 for an example of a heap-dependent function.

Methods A method has formal parameters and return parameters, pre- and postconditions that may contain accessibility predicates, and an optional body consisting of a sequence of statements. Examples of Viper methods can be found in listing 1.

Domains A domain in Viper can be used to encode mathematical theories. Each domain defines a potentially generic data type. The domain contains functions and axioms describing the theory. In contrast to functions outside a domain, domain functions are heap-independent. They only consist of a name, formal parameters and a type. Their behavior can be described using axioms. Note that axioms can only mention domain functions and not heap-dependent functions. They can however mention domain functions from any domain, not just their own. Like domain functions, axioms can only appear inside domains.

Inhale and Exhale A statement `inhale` e adds all the permissions in the assertion e, and assumes all pure parts of e. Analogously a statement `exhale` e asserts all pure parts of the assertion e and removes all the permissions in e. Viper also supports an *inhale-exhale* assertion $[A, B]$, that is interpreted as A in all inhaling positions (e.g. the postconditions of a method seen by the caller), and as B in all exhaling positions (e.g. the postconditions of a method seen by the method itself).

Conjunction The assertion a `&&` b corresponds to the separating conjunction `*` used in separation logic. It means a and b both hold and the permissions in a and the permissions in b do not overlap.

Magic Wand A magic wand A `--*` B describes a promise that if combined with a state satisfying assertion A, the combination can be exchanged for assertion B. Magic wands can be used to reduce the number of fold and unfold statements when manipulating data structures, since they provide a way to track partial versions of those data structures.

Permission Expressions The Viper expression `perm` $(x.f)$ yields the permission amount held for field f of reference x. The expression `forperm` $[f]\ r :: e(r)$ checks that for every reference r for which we have non-zero permission to the field f, $e(r)$ holds.

Triggers In Viper, forall assertions can have triggers:

`forall` $x_1 : T_1, \ldots, x_n : T_n :: \{t_{1,1}, \ldots, t_{1,m}\} \ldots \{t_{l,1}, \ldots, t_{l,m}\}\ e,$

where $\{t_{i,1}, \ldots, t_{i,m}\}$ is one trigger. Those triggers are given to the underlying SMT solver, where they are used for e-matching [6]. Viper generates default triggers if there are no user-defined triggers.

Natively Supported Types Viper supports the simple types integer, Boolean, permission, and reference. It also natively supports sets, sequences, and multisets.

2.2 GRASShopper

In the following paragraphs we explain the GRASShopper language features relevant to this report. Most of those features are described in more detail in [2], although the tool has changed since that paper was written. We describe some of those changes in this section, namely changes to predicates and functions.

Reference Types In GRASShopper, a reference type is defined as a struct with a list of fields. Getting permission to a location of some struct type, means getting access to all of its fields.

Permissions In contrast to Viper, GRASShopper permissions are not fractional, i.e. it only distinguishes between no permission and write permission. We use the accessibility predicate `acc` (x) to check that we have permission to reference x. If we have permission to a reference x, we are allowed to access *all* its fields.
Consider the following assertion:

```
acc(x) &*& (x.f > 0) &*& acc(y) &*& (z.f < 0)
```

We call the set `{x,y}` the *footprint* of this assertion. The footprint of an assertion contains all references the assertion checks permission to. The accessibility predicate `acc` (S) can also be used to check permission to a set of references S.

Checking Permissions In GRASShopper, `assert` statements, `assume` statements, and postconditions are checked in such a way that their footprint needs to correspond *precisely* to the currently held permissions. This means that in a context where we have permission to references `x` and `y`, the statement `assert acc(x)` will fail. The footprint of a loop invariant can be a subset of the current permissions at loop entry, but needs to correspond precisely to the permissions we hold at the end of the loop body.

Inside Assertion This binary assertion checks if the footprint of its left hand side is a subset the footprint of its right hand side. To check that e is inside E, we write e `-**` E.[2]

Set Comprehension GRASShopper supports a set comprehension expression

$$\{\, x : T :: b(x) \,\},$$

where $b(x)$ is a Boolean expression. An x is in the set iff $b(x)$ holds.

[2]Not to be confused with the magic wand symbol `--*` used in Viper.

```
struct Node {
  var next: Node;
}

predicate lseg(x: Node, y: Node) {
  acc({ z: Node :: Btwn(next, x, z, y) && z != y }) &*& Reach(next, x, y)
}

procedure traverse(lst: Node)
  requires lseg(lst, null)
  ensures lseg(lst, null)
{
  var curr := lst;
  while (curr != null)
    invariant lseg(lst, curr)
    invariant lseg(curr, null)
  {
    curr := curr.next;
  }
}
```

Listing 3: Definition of a linked list in GRASShopper using reachability predicates.

Predicates and Recursive Data Structures Instead of using recursive predicates, in GRASShopper one uses *reachability predicates* to describe recursive data structures such as linked lists. The predicate `Btwn` (f, a, b, c) expresses that node b lies between nodes a and c on a direct path always using field f to advance. `Reach` (f, a, b) is a shorthand for `Btwn` (f, a, b, b).
GRASShopper's reachability predicates can be used to describe linked list data structures as well as tree data structures. See listing 3 for an example of how to define a linked list segment in GRASShopper. Note that in comparison to the examples in [2] we do not need an extra function to define the footprint of the predicate anymore, instead we can specify the footprint directly inside the predicate using an accessibility predicate over a set comprehension. This is due to the recent changes in the GRASShopper tool.
In contrast to Viper, GRASShopper predicates do not need to be unfolded and folded using dedicated statements, because recursive predicates are not intended to be given to the internal GRASShopper verifier. There is no need to control the unfolding behavior of non-recursive predicates. As described in [2], in an earlier version of the GRASShopper tool there was an automatic translation of recursive predicates to non-recursive predicates that used reachability predicates instead. This translation was designed to work for definitions of linked lists. This functionality has been since removed from the tool. There are plans to implement a more general translation of recursive predicates to non-recursive predicates in the future. The current version of the GRASShopper tool has limited support for recursive predicates via axiomatization.

Functions GRASShopper functions may depend on the heap. In a recent version of the tool, support for specifying pre- and postconditions for functions was added, however those contracts are not checked yet.

Procedures GRASShopper procedures correspond to methods in Viper. An example of a procedure can be found in listing 3.

Axioms In contrast to Viper, axioms in GRASShopper can appear as top-level members of the program. GRASShopper axioms can mention heap-dependent functions, although one needs to explicitly quantify over the heap-dependent fields in the axiom.

Conjunction GRASShopper supports different kinds of conjunctions. a `&*&` b corresponds to the separating conjunction `*` in separation logic. a `&&` b corresponds to the logical conjunction $\wedge$ in separation logic. It means that the footprints of a and b have to be exactly the same. The conjunction a `&+&` b allows the footprints of a and b to overlap. If a and b are both pure, they need to be conjoined by `&&`.

Triggers In GRASShopper, triggers can be attached to forall expressions and set comprehensions. The triggers are divided into *matching* triggers, and *pattern* triggers. Matching triggers `@(matching` $t_1, \ldots, t_n$ `yields` e `)` work similarly to e-matching triggers. Both kinds of triggers are handled by GRASShopper directly, not the underlying SMT solver. Default triggers are always generated, user-defined triggers are added to the default triggers.

Natively Supported Types GRASShopper supports the simple types integer, Boolean, and byte[3]. It also natively supports sets, arrays, and uninterpreted data types.

[3]Needs to be enabled with a special flag.

3 Translating Viper to GRASShopper

In this section we demonstrate how to translate Viper language constructs to GRASShopper. In particular, we show how to translate constructs that do not have a direct correspondence in the GRASShopper language, such as inhale and exhale statements and quantified permissions. We also show how to handle language constructs that appear in both languages, but do not have exactly the same meaning in Viper and GRASShopper. An example for this category is our encoding of Viper fields that checks permissions per field, despite the fact that GRASShopper checks permissions per reference.

Notation We write the translation of some Viper assertion e as $[\![e]\!]$. The notation $e[e'/x]$ denotes the capture-avoiding replacement of all occurrences of x in the expression e with e'.

3.1 Folding and Unfolding Predicates

As discussed in section 2.2, GRASShopper does not have folding and unfolding operations. In our translation we simply drop `fold` and `unfold` statements. We translate an expression `unfolding` p `in` e to $[\![e]\!]$, essentially also dropping the unfolding operation.

3.2 Well-Formed Assertions

We say an assertion is *self-framing*, if it only accesses locations that it explicitly requires permission to, e.g. the assertion `acc(x.f) && x.f == 4` is self-framing, while the expression `x.f == 4` is not. An assertion is *well-formed*, if it only accesses locations that it has permission to, and is mathematically well-formed (e.g. no division by zero).
In Viper, checks need to be performed to make sure all assertions are well-formed, no matter where in the code they occur. GRASShopper only checks well-formedness for assertions that appear in `assert` statements, assignments and procedure calls. Since this behavior may change in the future, i.e. GRASShopper may implement well-formedness checks on all assertions, our translation does not add any such tests, which means that for contracts, predicate and function bodies, and axioms that are not well-formed, our tool will fail to report a corresponding error.

In our translation we often have to encapsulate assertions inside new GRASShopper predicates, functions, or procedures. Those encapsulated assertions again need to be well-formed (assuming the original assertions were well-formed), to make sure our translation still works once GRASShopper adds well-formedness checks.

3.3 Restrictions on Boolean Operators

Due to the internal handling of expressions in GRASShopper, Boolean operators are not allowed to appear in the following places:

- Arguments of predicates, functions, and procedures
- Right hand side of an assignment
- Triggers

In the category of Boolean operators, there are the following GRASShopper operators:

- General comparison operators `==` , `!=`
- Boolean operators `!` , `||` , `&&` , `&*&` , `&+&` , `==>`

- Integer comparisons `<`, `<=`, `>`, `>=`

In triggers, Viper does not allow Boolean operators either. Arguments and the right hand side of assignments are required to be pure expression in Viper. This lets us solve the problem by wrapping the Boolean operator inside an extra predicate. We can then replace the Boolean operator by an application of this predicate.

```
function t(j: Int): Int

method m(i: Int, s: Set[Int],
         b: Bool)
      returns (res: Bool)
{
  res := (b || t(i) < 0) &&
    forall x: Int :: x in s ==> x > 0
}
```

Viper

```
function t(j: Int) returns (res: Int)

procedure m(i: Int, s: Set<Int>,
            b: Bool)
      returns (res: Bool)
{
  res := bool_exp(b, t(i), s);

}

predicate bool_exp(b: Bool, v: Int,
                   s: Set<Int>)
{
  (b || v < 0) &&
    forall x: Int :: x in s ==> x > 0
}
```

GRASShopper

Translation 1: Encapsulation of Boolean operators

We explain our approach using translation 1 as an example. As arguments of the predicate application we take all subexpressions of the Boolean operator expression that are not Boolean operators themselves and that refer to a local variable available in the context where the Boolean operator expression appears. In our example those subexpressions are `b`, `t(i)`, and `s`. This excludes the local variable `x` of the forall expression inside the Boolean operator expression. We define the parameters of the new predicate according to the types of the extracted arguments. The body of the predicate becomes the translation of the original Boolean operator expression, but with the previously collected subexpressions replaced by the corresponding parameter names. This approach makes sure that the predicate body is self-framing, as long as the original expression itself was well-formed.

3.4 Conditional Assertions

A Viper conditional assertion has the following form:

$$e_{cond} \texttt{ ? } e_{then} \texttt{ : } e_{else},$$

where e_{cond} is a pure Boolean expression, and e_{then} and e_{else} are assertions of type T. GRASShopper does not support conditional assertions.

For cases where T is Boolean, we can translate the assertion to $(e_{cond} \implies e_{then}) \wedge (\neg e_{cond} \implies e_{else})$. If the whole conditional assertion is pure, this translates to

$$(\llbracket e_{cond} \rrbracket \texttt{ ==> } \llbracket e_{then} \rrbracket) \texttt{ \&\& } (\texttt{ ! } \llbracket e_{cond} \rrbracket \texttt{ ==> } \llbracket e_{else} \rrbracket).$$

In the general case we introduce an additional function $cond_T$, that takes parameters c : `Bool`, $a : \llbracket T \rrbracket$, and $b : \llbracket T \rrbracket$ and returns $x : \llbracket T \rrbracket$. The function has no body and a single postcondition:

$$(c \texttt{ ==> } (a \texttt{ == } x)) \texttt{ \&\& } (\texttt{ ! } c \texttt{ ==> } (b \texttt{ == } x))$$

Now, we can translate the conditional assertion as an application of $cond_T$ with arguments $[\![e_{cond}]\!]$, $[\![e_{then}]\!]$, and $[\![e_{else}]\!]$.

In both cases, if the assertion is non-pure, the `&&` needs to be replaced by `&*&` (see section 3.5).

3.5 Distinguishing between Assertions and Pure Expressions

As discussed in section 2.2 GRASShopper checks permissions *precisely* for `assert` statements, `assume` statements, postconditions, and loop invariants[4]. Assume we want to assert the pure expression `x.f == 5` in a context where we have permission to `x`. With precise permission checking we would have to write `assert x.f == 5 &*& acc(x)` to check this expression. To simplify checking of pure expressions, GRASShopper supports pure versions of all the checks, e.g. `pure assert`, `pure requires`, `pure ensures`, `pure invariant`. This simplifies our example to `pure assert x.f == 5`.
Whenever we translate a contract, `inhale` statement, or `exhale` statement, we check if the contained expression is pure. If it is indeed pure, we use the pure version of the check (e.g. `pure assert` for `exhale`, `pure assume` for `inhale`). The translation of the non-pure checks is usually more complicated. See sections 3.8, 3.9, and 3.12.

3.5.1 Boolean Conjunction

When translating a Boolean conjunction A `&&` B we have to distinguish whether the conjunction is pure or not, since GRASShopper distinguishes between separating conjunctions and logical conjunctions. If the conjunction is pure, we translate it to $[\![A]\!]$ `&&` $[\![B]\!]$, if the conjunction is not pure, we translate it to a separating conjunction $[\![A]\!]$ `&*&` $[\![B]\!]$.

3.6 Fields

In Viper, permissions are handled for each individual field of an object, while in GRASShopper, getting permission to an object includes permission to all its fields, i.e. GRASShopper does not support per-field permissions.
Consider the naïve encoding of Viper fields illustrated in translation 2. In this encoding we translate the Viper fields by putting them into a single GRASShopper struct, a field access $x.f$ is translated to $[\![x]\!].f$, while an accessibility predicate `acc` $(x.f)$ is translated to `acc` $([\![x]\!])$.
This naïve encoding has two problems:

- The encoding is not precise enough: `acc(x.f)` translates to `acc(x)`, which gives access to both `x.f` and `x.g` (see lines 6 and 8 of translation 2).
- The encoding is unsound: `acc(x.f) && acc(x.g)` translates to `acc(x) &*& acc(x)`, which evaluates to `false`, since the footprints of `acc(x)` and `acc(x)` overlap (see line 14 of translation 2).

With a more complex encoding of Viper fields in GRASShopper we can avoid those problems. The encoding is based on the idea that if we have a separate struct for each field, we can manage access to each original Viper field separately.
For this encoding we add a level of indirection:

- We use a `struct` R with no fields to represent the object itself.

[4]Loop invariants are checked precisely at the end of the loop, but their footprint can be a subset of the current permissions when entering the loop.

```
0
field f: Int
2 field g: Int


4
method m1(x: Ref)
6   requires acc(x.f)
{
8   x.g := 5 // should fail
}
10
method m2(x: Ref)
12  requires acc(x.f) && acc(x.g)
{
14  assert false // should fail
}
```

Viper

```
struct Node {
  var f: Int;
  var g: Int;
}

procedure m1(x: Node) returns ()
  requires acc(x)
{
  x.g := 5; // succeeds
}

procedure m2(x: Node) returns ()
  requires acc(x) &*& acc(x)
{
  pure assert false; // succeeds
}
```

GRASShopper

Translation 2: Naïve encoding of fields

- For each `field` $f : T$
  - we create a separate `struct` $struct_f$ with only a single field $f : [\![T]\!]$,
  - we add a `function` $map_f : R \to struct_f$, mapping objects of type R to objects of type $struct_f$.

Using this encoding, we can translate each field access $e.f$ to $map_f([\![e]\!]).f$, and each accessibility predicate `acc` $([\![e]\!].f)$ to `acc` $(map_f([\![e]\!]))$.
For this mapping to be correct, the mapping functions need to return a distinct reference for each pair of field f and input reference r. We already know that functions map_f and map_g return distinct sets of references, since they return objects of different types. All we need in addition is that every function map_f is injective. For assuming the injectivity of a function map_f, we have two possibilities:
Either we add an axiom

$$\forall x \in R : x \neq y \implies map_f(x) \neq map_f(x),$$

or we add another function $inv_f : struct_f \to R$ together with axioms stating that inv_f is the inverse of map_f

$$\forall x \in R : inv_f(map_f(x)) = x,$$

$$\forall y \in struct_f : map_f(inv_f(y)) = y.$$

In our implementation we use the second approach, since for the first approach the axiom needs to be instantiated for each pair of references, while for the second approach the axioms only need to be instantiated once for each reference. Another reason for using the second approach is that we need the inverse of map_f for the translation of quantified permissions (see section 3.12).

In listing 4 you can see the complete encoding for a single `field f: Int`. Note that the axioms have matching triggers to make sure that whenever map_f appears in the program, the prover learns about inv_f and vice versa. See section 3.10 for more detailed information on GRASShopper triggers.

```
struct R {}

struct struct_f {
  var f: Int;
}

function map_f(ref: R) returns (field: struct_f)
function inv_f(field: struct_f) returns (ref: R)

axiom forall x: R :: x == inv_f(map_f(x))
  @(matching map_f(y) yields known(inv_f(map_f(y))));

axiom forall z: struct_f :: z == map_f(inv_f(z))
  @(matching inv_f(z) yields known(map_f(inv_val(z))));
```

Listing 4: Encoding of field f

3.7 Domains

Our translation only supports domains without type parameters. We translate the type D defined by a domain `domain` D `{ ... }` to an uninterpreted data type in GRASShopper. Domain functions and axioms of the domain are translated to GRASShopper functions and axioms. See translation 3 for an example.

```
domain Pair {

  function pair(a: Int, b: Int): Pair

  function first(p: Pair): Int

  function second(p: Pair): Int


  axiom access_first {
    forall a: Int, b: Int ::
      first(pair(a, b)) == a
  }
  axiom access_second {
    forall a: Int, b: Int ::
      second(pair(a, b)) == b
  }
}
```

Viper

```
type Pair;

function pair(a: Int, b: Int)
  returns (result: Pair)
function first(p: Pair)
  returns (result: Int)
function second(p: Pair)
  returns (result: Int)

axiom forall a: Int, b: Int ::
  (first(pair(a, b)) == a);


axiom forall a: Int, b: Int ::
  (second(pair(a, b)) == b);
```

GRASShopper

Translation 3: Translation of a domain defining a pair of integers.

3.8 Precise Permission Checks

In Viper, a statement `assert acc(x.f)` checks that the specified permissions are in the current set of permissions, but in GRASShopper a statement `assert acc(x)` checks that the specified permissions correspond *precisely* to the current set of permissions. Similar restrictions also apply to postconditions, assumptions, and loop invariants.
Assume at some point in a Viper program we have permissions to `x.f` and `y.f`. A statement `assert acc(x.f)` would succeed in Viper. A naïve translation to GRASShopper, such as the statement `assert acc(`map_f`(x))` would fail in the GRASShopper tool, since

the footprint `acc(`map_f`(x))` does not correspond precisely to the current permissions.

In order to check an assertion in such a way that the assertion's footprint is allowed to correspond to a subset of the current permissions, we first need to obtain the current permissions, then we can use GRASShopper's inside assertion to express that the footprint of the translated expression should be a subset of the current permissions.

Getting the Current Set of Permissions Let's assume there are structs A and B in some GRASShopper program. In a first step, we define a fresh set variable for each struct:

`var` $perms_A$ `: Set<`A`>,` $perms_B$ `: Set<`B`> ;`

In the next step we bind those sets to the set of references we currently have permission to. Since assumptions in GRASShopper are checked against the current permissions precisely, we can just assume that sets $perms_A$ and $perms_B$ contain exactly the objects we currently have permission to:

`assume acc(`$perms_A$`) &*& acc(`$perms_B$`) ;`

In the following we define $Perms$ to be the set of all the required set variables to contain the permissions for some program. In our example above this would be $\{perms_A$: `Set<`A`>`, $perms_B$: `Set<`B`>`$\}$. $Perms(s)$ denotes the single variable in $Perms$ that is of struct type s. In our example $Perms(A)$ would be $perms_A$. By $acc(Perms)$ we denote the separating conjunction over accessibility predicates over the set variables, e.g.

`acc(`$perms_A$`) &*& acc(`$perms_B$`)`.

In the following, we discuss how to use the current set of permissions for translating Viper assertions, postconditions, and loop invariants in such a way that their permissions are allowed to be a subset of the current permissions.

Assert Statements The translation of an assert statement first gets the current permissions $Perms$, as discussed above. Now we can translate `assert` e to

`assert` $[\![e]\!]$ `-**` $acc(Perms)$.

The inside assertion used here checks that $[\![e]\!]$ holds and its footprint is a subset of $acc(Perms)$. The footprint of $[\![e]\!]$ `-**` $acc(Perms)$ corresponds exactly to the current permissions, meaning GRASShopper's precise permission checking won't reject it.

Postconditions If there is a non-pure postcondition in the method, we check all the postconditions manually. First, we conjoin all the postconditions using separating conjunction. Then we exhale[5] the conjunction (see section 3.9.2). Our exhale translation takes care of checking the permissions to be inside the current set of permissions. To keep the interface to the procedure unchanged, we translate the actual postconditions of the method directly. To make sure the GRASShopper tool can prove those postconditions, we assume `false` at the end of the procedure body.

Loop Invariants GRASShopper checks that the footprint of an invariant corresponds to a subset of the current permissions at loop entry, but checks that the invariant's footprint corresponds precisely to the current permissions at the end of the loop body. Therefore, if there is a non-pure loop invariant in the definition of a loop, we translate the loop without

[5]Actually, we only perform phases 1 and 2 of the exhale. This only checks if we *could* do the exhale, without actually giving the permissions away. See section 3.9.2 for details.

any invariants, but add manual checks instead. We exhale the invariants before the loop and inhale them after the loop. At the beginning of the loop body we inhale the invariants, at the end of the loop body we exhale the invariants again. See section 3.9.3 for details.

3.9 Inhale and Exhale

GRASShopper does not support inhale or exhale statements, but we can simulate removing and adding permissions by calling a procedure that only has a postcondition or precondition, respectively. See translation 4 for an example.

```
method m(x: Ref)
{
  inhale acc(x.f)
  exhale acc(x.f)
}
```

Viper

```
procedure m(x: R)
{
  inhale(x);
  exhale(x);
}

procedure inhale(x: R)
  ensures acc(map_f(x))

procedure exhale(x: R)
  requires acc(map_f(x))
```

GRASShopper

Translation 4: Adding and removing permissions in GRASShopper

3.9.1 Inhale

A naïve translation of the inhale statement to a single procedure call would be incorrect, consider the following example:

Assume that we have an inhale statement with a mixture of pure expressions and assertions such as this one:

```
inhale x.f == 4 && acc(x.g)
```

Using the assertion `x.f == 4 && acc(x.g)` as a postcondition of an extra GRASShopper procedure would be incorrect, since the postcondition would access `x.f`, but does not have permission to it. Figuring out which permissions we would need to add as precondition would require a detailed analysis of assertions. Instead we break up the assertion parameter to the inhale into subexpressions that can each be inhaled separately. In appendix A we describe how to break apart the assertion. In the following we describe how to handle the base cases.

- A pure expression e is translated into an assertion statement: `pure assert` $[\![e]\!]$
- `acc` $(e.f)$ is translated to a procedure call $m_{inhale}([\![e]\!])$, where the new procedure m_{inhale} has a single parameter $a : T$ corresponding to $[\![e]\!]$, and a single postcondition `acc` $(map_f(a))$.
- `acc` $(p(e_1, ..., e_n))$ is translated to a procedure call $m_{inhale}([\![e_1]\!], ..., [\![e_n]\!])$, has parameters $a_1 : T_1, ..., a_n : T_n$ corresponding to $[\![e_1]\!], ..., [\![e_n]\!]$ and the single postcondition `acc` $(p(a_1, ..., a_n))$.
- We discuss how to handle quantified permissions in inhale statements in section 3.12.2.

3.9.2 Exhale

There are the following constraints on how the exhale statement needs to behave:

- Pure expressions need to be checked against the permissions from *before* the exhale statement.
- Assertions need to be checked against the *current* permissions as they are modified by the exhale.

One possibility to satisfy these constraints would be to reorder the checks inside the exhale, so that the pure expressions are checked first and the permissions are exhaled in the end. Using this approach however, we may not report the expected errors, because some errors may be shadowed by others. By reordering the checks we change this shadowing behavior. In our implementation, we use a more complex approach, that preserves the order of the checks. We check pure expressions against the unchanged permissions, and assertions against sets representing the permissions as they are affected by the exhale. Only at the end of the exhale we give away the permissions. This approach is structured into three phases:

Phase 1 Get the current permissions $Perms_{curr}$, this will represent the permissions as they are modified by the exhale. Initialize empty sets $Perms_{ex}$ to gather permissions that need to be exhaled.

Phase 2 Assert pure parts of the assertion, check non-pure parts of the assertion against $Perms_{curr}$, remove permissions from $Perms_{curr}$ and add them to $Perms_{ex}$.

Phase 3 Remove all the permissions in $Perms_{ex}$.

We will now discuss phases 2 and 3 in more detail. For phase 1 see section 3.5 on how to get the current permissions.

In phase 2, the assertion is broken up into multiple statements as discussed in appendix A. For the non-pure base cases, we assert that their footprint is in $Perms_{curr}$, remove their footprint from $Perms_{curr}$ and add it to $Perms_{ex}$:

- `acc` $(e.f)$: The footprint of this assertion is $map_f([\![e]\!])$. We check if this is in $Perms_{curr}$:

  `pure assert` $map_f([\![e]\!])$ `in` $Perms_{curr}(struct_f)$ `;`

  Then we remove it from $Perms_{curr}$ and add it to $Perms_{ex}$:

  $Perms_{curr}(struct_f)$ `:=` $Perms_{curr}(struct_f)$ `--` $map_f([\![e]\!])$ `;`
  $Perms_{ex}(struct_f)$ `:=` $Perms_{ex}(struct_f)$ `++` $map_f([\![e]\!])$ `;`

  Note that `--` and `++` are GRASShopper's set minus and set union, respectively.

- `acc` $(p(e_1, ..., e_n)$: The footprint of a predicate is not known at translation time. First we get the actual current permissions $Perms_{before}$[6], since we will have to write some assert and assume statements for which we need to know the current permissions. The permissions saved in $Perms_{curr}$ may not correspond to the current permissions anymore, since $Perms_{curr}$ may have been modified when handling earlier parts of the exhale assertion. In the next step, we define a set of fresh variables $Perms_p$, which will hold the footprint of the predicate. Then we assert that the footprint of the predicate is inside the current permissions:

  `assert` $[\![p(e_1, ..., e_n)]\!]$ `-**` $acc(Perms_{before})$ `;`

[6]The current permissions are still the same as from before the exhale statement.

```
... // Field translations

procedure m(x: R)
{
  // Phase 1
  var Perms_struct_f: Set<struct_f>, Perms_R: Set<R>;
  assume (acc(Perms_struct_f) &*& acc(Perms_R));
  var Perms_struct_f_1: Set<struct_f>, Perms_R_1: Set<R>;
  assume (acc(Perms_struct_f_1) &*& acc(Perms_R_1));
  var ex_Perms_struct_f: Set<struct_f>, ex_Perms_R: Set<R>;
  ex_Perms_struct_f := Set<struct_f>();
  ex_Perms_R := Set<R>();

  // Phase 2
  ...

  // Phase 3
  pExhale(ex_Perms_struct_f, ex_Perms_R);
}

procedure pExhale(Perms_struct_f: Set<struct_f>, Perms_R: Set<R>)
  requires (acc(Perms_struct_f) &*& acc(Perms_R))
```

Listing 5: Phases 1 and 3 of an exhale in a program with a single field f

Now, we can write

`assume` $([\![p(e_1, ..., e_n)]\!]$ `&&` $acc(Perms_p))$ `-**` $acc(Perms_{before})$ `;`

Here, the conjunction $[\![p(e_1, ..., e_n)]\!]$ `&&` $acc(Perms_p)$ sets the footprint of $acc(Perms_p)$ to be equal to the footprint of the predicate application, the inside assertion makes sure that the footprint of the whole assertion corresponds exactly to the current permissions. This means, after the assume statement $Perms_p$ contains the footprint of $[\![p(e_1, ..., e_n)]\!]$.
Since we don't know of what type the footprint of $[\![p(e_1, ..., e_n)]\!]$ is, we need to handle each struct s individually. For every struct s in the program we check that $Perms_p(s)$ is in $Perms_{curr}(s)$, remove it from there, and add it to $Perms_{ex}(s)$:

`pure assert` $Perms_p(s)$ `subsetof` $Perms_{curr}(s)$ `;`
$Perms_{curr}(s)$ `:=` $Perms_{curr}(s)$ `--` $Perms_p(s)$ `;`
$Perms_{ex}(s)$ `:=` $Perms_{ex}(s)$ `++` $Perms_p(s)$ `;`

- We discuss how to handle quantified permissions in exhale statements in section 3.12.3.

The handling of a pure expression e is much simpler. We first assert $[\![e]\!]$ and then assume it:

`pure assert` $[\![e]\!]$ `;`
`pure assume` $[\![e]\!]$ `;`

Using a pair of assert and assume statements emulates the behavior of Viper in case of multiple errors in a single method.

In phase 3, we need to exhale all the permissions that are now stored in $Perms_{ex}$. For this we introduce an extra procedure m_{exhale} with the variables in $Perms_{ex}$ as parameters and the single precondition $acc(Perms_{ex})$. To exhale the permissions in $Perms_{ex}$ we call m_{exhale} with the variables of $Perms_{ex}$ as arguments. See listing 5 for an example on how phases 1 and 3 of the exhale are encoded.

3.9.3 Inhale-Exhale Assertion

A Viper inhale-exhale assertion $[A, B]$ is considered non-pure, and may not appear in positions where only pure expressions are allowed (e.g. conditions of if statements). In the following, we discuss how to handle the inhale-exhale assertion, depending on where it appears.

Assert Statement `assert` $[A, B]$ is equivalent to `assert` B.

Inhale `inhale` $[A, B]$ is equivalent to `inhale` A.

Exhale `exhale` $[A, B]$ is equivalent to `exhale` B.

Whenever we encounter one of the above three statements, we can transform the original assertion to an assertion with the inhaling-exhaling assertions replaced by the correct subexpression. Then we can translate it as we would any such statement.

Method Pre- and Postconditions Preconditions get *exhaled* by the caller, and *inhaled* by the callee, while postconditions get *inhaled* by the caller, and *exhaled* by the callee.
In order to keep the interface to the method the same, we translate a Viper precondition `requires` $[pre_{in}, pre_{ex}]$ to a GRASShopper precondition `requires` $[\![pre_{ex}]\!]$ and a Viper postcondition `ensures` $[post_{in}, post_{ex}]$ to a GRASShopper postcondition `ensures` $[\![post_{in}]\!]$.
Now we need to adapt the method body such that the correct checks are performed. The body needs to start in a state where $[\![pre_{in}]\!]$ holds, after the body we need to check $[\![post_{ex}]\!]$ holds, but the GRASShopper tool needs to be able to prove the postcondition $[\![post_{in}]\!]$.
To achieve this behavior we exhale $[\![pre_{ex}]\!]$ and inhale $[\![pre_{in}]\!]$ before the actual method body using the techniques discussed previously. We can drop the exhale if pre_{ex} is a pure expression, since exhaling it would be equivalent to asserting it, which would be redundant.
After the method body, we perform phases 1 and 2 of an exhale on $post_{ex}$, and then we assume `false`. This lets us prove the postcondition $[\![post_{in}]\!]$. We don't need phase 3 of the exhale, since we don't really need to give away the permissions, we only need to check if we *could* give them away. See section 3.9.2 for a definition of the exhale phases.
In translation 5 you can see an example of how inhale-exhale assertions in method pre- and postconditions are handled.

Loop Invariant If any loop invariant contains an inhale-exhale assertion, we generate code to check the invariants ourselves and do not supply the GRASShopper loop with any invariants. The statements we generate are slightly different depending on whether the loop condition depends on the heap or not. We first discuss how to solve the problem if the condition is not heap-dependent, and then explain how to modify this solution such that it works for heap-dependent conditions.

Suppose this is our while statement: `while` (b) `invariant` $[A, B]$ `{` *body* `}`
If we assume that b is *not* heap-dependent, then the whole while statement is translated as follows:

- Before the translation of the while statement, exhale B.
- Inside the loop body, before the translation of the actual body, inhale A.
- Inside the loop body, after the translation of the actual body, exhale B.

```
method m(a: Ref, b: Ref)
  requires [acc(a.f), acc(a.g)]
  ensures [acc(b.f), acc(b.g)]
{
  // body
}
```

Viper

```
procedure m(a: R, b: R)
  requires acc(map_g(a))
  ensures acc(map_f(b))
{
  // exhale acc(map_g(a))
  // phase 1
  var Pm_f: Set<struct_f>, ...;
  assume acc(Pm_f) &*& ...;
  var ex_Pm_f: Set<struct_f>, ...;
  ex_Pm_f := Set<struct_f>();
  ...
  // phase 2
  pure assert map_g(a) in Pm_g;
  Pm_g := Pm_g -- {map_g(a)};
  ex_Pm_g := ex_Pm_g ++ {map_g(a)};
  // phase 3
  exhale(ex_Pm_f, ex_Pm_g, ex_Pm_R);

  // inhale acc(map_f(a))
  inhale(a);

  // body

  // "exhale" acc(map_g(b))
  // phase 1
  var Pm_f2: Set<struct_f>, ...;
  assume acc(Pm_f2) &*& ...;
  var ex_Pm_f1: Set<struct_f>, ...;
  ex_Pm_f1 := Set<struct_f>();
  ...
  // phase 2
  pure assert map_g(b) in Pm_g2;
  Pm_g2 := Pm_g2 -- {map_g(b)};
  ex_Pm_g1 := ex_Pm_g1 ++ {map_g(b)};

  // kill branch
  assume false;
}

procedure exhale(Pm_struct_f: Set...)
  requires acc(Pm_struct_f) &*& ...

procedure inhale(a: R)
  ensures acc(map_f(a))
```

GRASShopper

Translation 5: Inhale-exhale assertions in method pre- and postconditions.

- After the while statement, inhale A.
- The translation of the while statement has no invariant.

If b is *heap-dependent* we need to make the following changes:

- Before the translation of the while statement (and before our exhale), introduce an additional variable c : `Bool` and assign $[\![b]\!]$ to it: $c := [\![b]\!]$
- Use c as the condition of the while statement instead of $[\![b]\!]$
- Inside the loop body, before the translation of the actual body, additionally assume $c == [\![b]\!]$.
- Inside the loop body, after the translation of the actual body, assign $[\![b]\!]$ to c again, before exhaling B.
- After the while statement, additionally assume $c == [\![b]\!]$.

Others We don't support inhale-exhale assertions in function pre- and postconditions, or in predicate bodies.

3.10 Pure Quantified Expressions

Existentially quantified expressions are trivial to translate from Viper to GRASShopper:

`exists` $x_1 : T_1, \ldots x_n : T_n :: e(x_1, \ldots, x_n)$

is translated to

`exists` $[\![x_1 : T_1]\!], \ldots [\![x_n : T_n]\!] :: [\![e(x_1, \ldots, x_n)]\!]$.

Translating universally quantified pure expressions is a little bit more complex, since those expressions can have triggers. GRASShopper generates its own default triggers, therefore we don't have to generate triggers when translating a Viper forall expressions without user-defined triggers:

`forall` $x_1 : T_1, \ldots x_n : T_n :: e(x_1, \ldots, x_n)$

is translated to

`forall` $[\![x_1 : T_1]\!], \ldots [\![x_n : T_n]\!] :: [\![e(x_1, \ldots, x_n)]\!]$.

A forall expression with triggers has the following form:

`forall` $x_1 : T_1, \ldots, x_n : T_n :: \{t_{1,1}, \ldots, t_{1,m}\} \ldots \{t_{l,1}, \ldots, t_{l,m}\} e(x_1, \ldots, x_n)$.

A triggering expression $t_{i,j}$ is given in terms of the variables in $x_1, \ldots, x_n$. The triggering expressions are only allowed to be function applications, or expressions that are translated to function applications internally, e.g. set operators.

We translate the Viper triggers to matching triggers in GRASShopper, since these work similarly to the e-matching triggers used by the other backends of Viper. Matching triggers have the following form:

`@(matching` $m_1, \ldots, m_n$ `yields` e' `)`

Expressions $m_1, \ldots, m_n, e'$ may not contain Boolean operators (see section 3.3). Matching known expressions $m_1, \ldots, m_n$ adds e' to the set of known expressions, but e' itself is not used in the instantiation of new known expressions, unless it is specified as `known` (e'). Using triggers with a known expression in the yield position emulates the behavior of e-matching, where every generated term can itself be used for triggering again.

Trigger Translation We collect all subexpressions E of e that are valid *Viper* triggers. For each pair of subexpression $e' \in E$ and Viper trigger $\{t_{i,1}, \ldots, t_{i,n}\}$ we add a matching trigger to the translation of the forall expression.

@(matching $[\![\{t_{i,1}, \ldots, t_{i,n}\}]\!]$ yields known($[\![e']\!]$))

3.11 Permission Expressions

Viper supports fractional permissions, while in GRASShopper we can only distinguish between write permission, and no permission at all. Therefore in our translation, we do not support fractional permissions. However, there are some permission expressions that are useful, even without fractional permissions. The conditional expression as discussed in section 3.4 is also applicable for expressions of type `Perm`: b ? p_{then} : p_{else}.

Algorithm 1 Generate condition under which we have full permission from permission expression

```
1: function FullPermissionCondition(p)
2:     switch p do
3:         case write
4:             return true
5:         case none
6:             return false
7:         case cond ? p_then : p_else
8:             return ([[e_cond]] && FullPermissionCondition(p_then))
9:              || ( ! [[e_cond]] && FullPermissionCondition(p_else))
```

Our translation supports such conditional permission expressions, together with the full permission `write`, and the expression denoting no permission `none`, but only inside field access predicates `acc` $(e.f, p)$. We translate the permission expression p into a condition under which the access predicate requires full permission as shown in algorithm 1. Let's call the result of this algorithm c_p.
We translate the field access predicate as follows: If c_p is the literal `true` we translate the access predicate to `acc` $(map_f([\![e]\!]))$; this represents the full permission. If c_p is the literal `false` we translate the access predicate to `true`, representing no permission. If c_p is a more complex expression, we translate it to c_p `==>` `acc` $(map_f([\![e]\!]))$, representing a permission under the condition c_p.
When translating inhale and exhale statements, every field access predicate is translated to a procedure call as discussed in section 3.9. In this context, a more complex c_p needs to be translated to an if statement rather than an implication, to make sure that the postconditions of our extra inhale procedures stay well-formed:

```
if (c_p) {
  // inhale acc (map_f([[e]]))
}
```

3.12 Quantified Permissions

Viper allows only one type of non-pure forall assertion at the moment. This so-called *quantified permission* assertion needs to have the following form:

$$\texttt{forall}\ x : T :: b(x) \texttt{ ==> } \texttt{acc}\,(e(x).f,\ p(x)),$$

where $b(x)$ is a pure Boolean expression, $e(x)$ is an expression evaluating to a reference, f is some field, and $p(x)$ is a permission expression. We cannot directly translate this assertion to GRASShopper, since GRASShopper only supports pure forall expressions.

P. Müller, M. Schwerhoff, and A. J. Summers describe in [5] how we can translate quantified permissions from a forall assertion over T into a forall assertion in terms of `Ref`. Our translation uses a modified approach based on their idea, in order to translate the quantified permission assertion to a set comprehension over $struct_f$, containing all locations the quantified permission assertion requires access to.

Viper expects that the expression e is asserted to be injective as part of the well-formedness checks. We check if e is injective, under the assumptions that b holds and we are given enough permissions:

$$\forall y, z : T ::$$
$$y \neq z \wedge b[y/x] \wedge b[z/x] \wedge p[y/x] > \texttt{none} \wedge p[z/x] > \texttt{none} \implies e[y/x] \neq e[z/x]$$

Since we do not support fractional permissions in our translation, this expression can be simplified to:

$$\forall y, z : T ::$$
$$y \neq z \wedge b[y/x] \wedge b[z/x] \wedge p[y/x] = \texttt{write} \wedge p[z/x] = \texttt{write} \implies e[y/x] \neq e[z/x].$$

Under the assumption that e is injective, we can define an inverse function inv_e for the expression e:

$$(\forall x : T :: b \implies inv_e(e) = x) \wedge (\forall w : \texttt{Ref} :: b[inv_e(w)/x] \implies e[inv_e(w)/x] = w)$$

Now we can write the quantified permission assertion as a forall assertion in terms of references:

$$\forall r : \texttt{Ref} :: b[inv_e(r)/x] \wedge (p[inv_e(r)/x] = \texttt{write}) \implies \texttt{acc}\,(inv_e(r).f)$$

We will now show how those conditions and assumptions can be expressed in GRASShopper. Using the condition under which we have full permission c_p (see section 3.11), the translation of the injectivity test is straightforward:

$$\texttt{forall}\ y : [\![T]\!], z : [\![T]\!] ::$$
$$y\ \texttt{!=}\ z\ \texttt{\&\&}\ [\![b]\!][y/[\![x]\!]]\ \texttt{\&\&}\ [\![b]\!][z/[\![x]\!]]\ \texttt{\&\&}\ c_p[y/[\![x]\!]]\ \texttt{\&\&}\ c_p[z/[\![x]\!]]\ \texttt{==>}\ [\![e]\!][y/[\![x]\!]]\ \texttt{!=}\ [\![e]\!][z/[\![x]\!]]$$

Next, we introduce a new function $inv_e : T \to R$ and add assumptions stating that inv_e is the inverse of e:

$$\texttt{forall}\ [\![x : T]\!] ::\ [\![b]\!] \texttt{==>}\ inv_e([\![e]\!])\ \texttt{==}\ [\![x]\!]$$
$$\texttt{forall}\ w : R ::\ [\![b]\!][inv_e(w)/[\![x]\!]] \texttt{==>}\ [\![e]\!][inv_e(w)/[\![x]\!]]\ \texttt{==}\ w$$

The forall expression from above is in terms of `Ref` (which would be translated to type R as defined in section 3.6), but we need the set comprehension to be in terms of $struct_f$. So we replace r : `Ref` by $inv_f(r_f) : R$, where $r_f : struct_f$. Now we can define the set comprehension as follows:

$$\texttt{acc}\,(\{\ r_f : struct_f ::\ [\![b]\!][inv_e(inv_f(r_f))/[\![x]\!]]\ \texttt{\&\&}\ c_p[inv_e(inv_f(r_f))/[\![x]\!]]\ \})$$

In the case where e is just the expression x, we don't need the injectivity test or the definition of the inverse function, since x is trivially injective, i.e. the inverse function is the identity. This simplifies the set comprehension as follows:

$$\texttt{acc}\,(\{\ r_f : struct_f ::\ [\![b]\!][inv_f(r_f)/[\![x]\!]]\ \texttt{\&\&}\ c_p[inv_f(r_f)/[\![x]\!]]\ \})$$

This is an optimization that is not yet implemented in the other Viper backends. Using this

optimization would simplify the generated verifier code and make it more readable, which is already the case in our translation.

In the upcoming part of this section we discuss how to use the injectivity test, inverse assumptions, and set comprehension to translate a quantified permission assertion occurring in different positions. In all those cases we concentrate on the case where e is not the expression x. In the case where e is the expression x we translate a quantified permission assertion to an accessibility predicate over the simplified set comprehension, regardless of the location.

3.12.1 Assert Statement

Since we need to *assert* that e is injective, but *assume* that inv_e is the inverse of e, we cannot translate a Viper assert statement containing a quantified permission assertion to a single GRASShopper assert statement. Therefore, we break up the assertion of the assert statement into subexpressions, as we do e.g. with the inhale statement in section 3.9.1 (see more details in appendix A), but with a small change: the criterion for whether to break the assertion up further is whether the assertion contains a quantified permission assertion. The base cases are the quantified permission assertions themselves, and subexpressions without any quantified permissions.
The quantified permission assertions are translated to a sequence of statements. First we get the current permissions *Perms* as described in section 3.8, then we assert that e is injective and assume the existence of an inverse function. Then we can add the actual assert statement using the set comprehension from above.

```
// get current permissions
pure assert forall y : [[T]], z : [[T]] ::
    y != z && [[b]][y/[[x]]] && [[b]][z/[[x]]] && c_p[y/[[x]]] && c_p[z/[[x]]]
        ==> [[e]][y/[[x]]] != [[e]][z/[[x]]] ;
pure assume forall [[x : T]] :: [[b]] ==> inv_e([[e]]) == [[x]] ;
pure assume forall w : R :: [[b]][inv_e(w)/[[x]]] ==> [[e]][inv_e(w)/[[x]]] == w ;
assert acc({ r_f : struct_f :: [[b]][inv_e(inv_f(r_f))/[[x]]] && c_p[inv_e(inv_f(r_f))/[[x]]] })
    -** acc(Perms) ;
```

3.12.2 Inhale

Since the quantified permission assertion is non-pure, an inhale containing a quantified permission assertion is already being broken up into multiple statements (see section 3.9.1). Here we discuss how to handle the quantified permission assertion as a base case when breaking up the inhale statement. As with the assert statement, we first assert that e is injective. To add the permissions in the quantified permission assertion we create another procedure m_{inhale}, with the assumption that inv_e is the inverse of e and the accessibility predicate over the set comprehension as its postconditions. As the arguments of the procedure we use all the local variables mentioned, but not defined in the quantified permission assertion.
This approach only guarantees that the postconditions are well-formed if the quantified permission assertion was *self-framing*. Consider the following quantified permission assertion:

```
forall x: T :: y.g ==> acc(fun(x, y).f),
```

where `y` is some reference we have permission to in the local context, and `fun` is some function. Our postcondition would need permission to `y.g` and potentially more fields of `y` to satisfy the precondition of `fun`. Computing exactly which permissions would be needed

is subject to future work. Definition 10 in [7] defines when a set of permissions is enough so that an assertion is well-formed. This notion could be used to write an algorithm that finds the smallest set of permissions, such that a given assertion is well-formed.

3.12.3 Exhale

For handling an exhale statement containing a quantified permission assertion we add another base case to phase 2 of the exhale (see section 3.9.2). We add an assert statement for checking the injectivity and assume statements for the inverse function. Then we remove the set of permissions from $Perms_{curr}$ (current permissions) and add them to $Perms_{ex}$ (permissions to exhale):

$Perms_{curr}(struct_f)$:= $Perms_{curr}(struct_f)$
    -- { $r_f : struct_f$ :: $[\![b]\!][inv_e(inv_f(r_f))/[\![x]\!]]$ && $c_p[inv_e(inv_f(r_f))/[\![x]\!]]$ };
$Perms_{ex}(struct_f)$:= $Perms_{ex}(struct_f)$
    ++ { $r_f : struct_f$ :: $[\![b]\!][inv_e(inv_f(r_f))/[\![x]\!]]$ && $c_p[inv_e(inv_f(r_f))/[\![x]\!]]$ };

3.12.4 Method Postcondition

If a method postcondition contains a quantified permission assertion, then all the postconditions are conjoined and handled like an exhale statement at the end of the method. This is done because GRASShopper checks that the footprint of the postconditions corresponds exactly to the permission at the end of the procedure body, while in Viper the footprint of the postconditions is allowed to be a subset of the permissions at the and of the method body (see section 3.8). The *checking* of the postcondition is therefore handled like in an exhale (see section 3.12.3) of the quantified permission assertion.
The postcondition a method call gets to *assume* (i.e. the actual postcondition of the translated method) is the separating conjunction of the assumptions for the inverse function and the accessibility predicate over the set comprehension. This means, injectivity of e gets assumed, rather than tested at caller site. This is a different behavior than expected in Viper, where injectivity also needs to be checked when inhaling method postconditions.

3.12.5 Method Precondition

If a method precondition contains a quantified permission, we need to check the preconditions at call site. This is because the preconditions are handled as assertions at call site, but we need to *assume* that there exists an inverse function, but only after we checked injectivity.
We solve this problem by omitting the method call entirely; instead we exhale the translated preconditions of the method and inhale its postconditions, using the techniques discussed in section 3.9. How quantified permissions are handled in exhales is described in section 3.12.3.
In the actual method precondition, we translate the quantified permission assertion into the separating conjunction of the inverse assumption and the accessibility predicate of the set comprehension. As with the method postconditions, this is a different behavior than expected in Viper, where injectivity also needs to be checked when inhaling method preconditions.

3.12.6 Loop Invariant

If a loop invariant contains a quantified permission assertion it is a non-pure invariant. If any invariant of a loop is non-pure, we add manual checks for the loop invariant, and don't supply the GRASShopper loop with any invariants (see section 3.8). This means, the loop invariant containing the quantified permission will be only used in positions where it will either be

handled like an inhale or an exhale statement. How to handle the quantified permission assertion in those situations we already discussed in sections 3.12.2 and 3.12.3.

3.12.7 Others

In any other positions our translation defaults to a separating conjunction of inverse assumptions and the accessibility predicate over the set comprehension. Those positions include function preconditions, function bodies, and predicate bodies. This default handling does not correspond to the expected Viper behavior of the affected functions and predicates.

4 Translating GRASShopper to Viper

In addition to our translation from Viper to GRASShopper we have laid the foundations of a translation from GRASShopper to Viper. Our translation assumes that the supplied GRASShopper program is syntactically correct and properly typed, i.e. the program is accepted by the GRASShopper tool's type checker. We can run the GRASShopper tool such that it stops after type checking by specifying the option `-typeonly`. Our translation tool does not implement type inference, but assumes that no type inference has to be performed on the input program. Running the GRASShopper tool with the option `-simplify`[7] will print out a version of the original GRASShopper program augmented with the inferred type information. In this new version of the GRASShopper program an expression `{1, 2, 3}` will be replaced by `Set<Int>(1, 2, 3)` and a statement `var b := false;` will be replaced by statements `var b: Bool; b := false;`.
Our simple translation from GRASShopper to Viper supports all the language features of GRASShopper that we use in our translation from Viper to GRASShopper. However, for the translation of some of those language features we make simplifying assumptions that do not necessarily hold in every GRASShopper program. In the following, we describe which language features of GRASShopper our translation supports, how those language features are translated, and what the simplifying assumptions are.

Structs We translate all GRASShopper struct types to the `Ref` type in Viper, and translate each struct definition to a sequence of field definitions. Since GRASShopper manages permissions per reference, but Viper manages permissions per pair of field and reference, we translate an accessibility predicate `acc`(x) of a single reference x to the conjunction of accessibility predicates for all the fields corresponding to the type of x. Similarly, we translate creating a new object to creating a new reference with permissions to the fields corresponding to the struct's fields. See translation 6 for an example.

```
struct A {
  var f: Int;
  var g: A;
}

procedure m(a: A)
  requires acc(a)
{
  var b: A;
  b := new A;
}
```

GRASShopper

```


field f: Int
field g: Ref


method m(a: Ref)
  requires acc(a.f) && acc(a.g)
{
  var b: Ref
  b := new(f, g)
}
```

Viper

Translation 6: Translation of GRASShopper structs

We translate an accessibility predicate `acc`(S) over a set S : `Set<`T`>` to the conjunction of quantified permission assertions for each field f of struct T:

`forall` x : `Ref` :: x `in` S `==>` `acc`$(x.f)$

Checking Assertions As discussed in section 3.8, permissions occurring in assertions, assumptions, loop invariants, and postconditions are checked differently in GRASShopper than

[7] This feature is under development at the time of writing this report.

in Viper. Viper checks that the mentioned permissions are a subset of the currently held permissions, while GRASShopper checks that the mentioned permissions correspond precisely to the currently held permissions. Our translation currently ignores that fact and translates assertions, loop invariants, and postconditions to Viper without modification. This means we effectively weaken all the permission checks.
As described in section 3.5, GRASShopper supports pure versions of all checks. By directly translating those pure checks to the corresponding checks in Viper, we do not weaken the checks, since the pure checks can only mention pure expressions. The `pure assert` statement is translated to an `inhale` statement in Viper.

Assume Statement and Inside Expression In our translation from Viper to GRASShopper, the use of assume statements and inside assertions is connected to circumventing GRASShopper's precise permission checking. In our translation from GRASShopper to Viper we drop `assume` statements and translate inside assertions A `-**` B to the translation of just A.
We justify this approach with the following example: Assume we translate `assert` C from Viper to GRASShopper. According to section 3.8, this statement is translated to first obtaining the current permissions *Perms* using an assume statement over $acc(Perms)$, and then asserting $[\![C]\!]$ `-**` $acc(Perms)$. Our translation back to GRASShopper will then drop the assumption, and only translate back the definitions of the variables in *Perms* and an assert statement of $[\![C]\!]$. After being translated to GRASShopper and then back to Viper again, this assertion will verify in the same way as the original Viper assertion.
However, we also generate `assume` statements in our translation of Viper `exhale` statements (see section 3.9.2). Here, the obtained current permissions *Perms* are modified using set operations which influence the behavior of the exhale. For each of the exhale's subexpressions that mention permissions, we check if the corresponding object is in *Perms* using a set inclusion operation. This operation will fail in our translation from GRASShopper to Viper, since we dropped the assumption that *Perms* contains the currently held permissions. To solve this problem, the translation from GRASShopper to Viper could be improved in the following way: Instead of dropping all `assume` statements, we would translate those `assume` statements that obtain the set of current permissions using Viper's `forperm` expression. Consider the following assume statement:

`assume acc` (S_1) `&*&  acc` (S_2),

where S_1 and S_2 are sets of different struct types T_1 and T_2. The fields of all structs in GRASShopper are distinct, therefore we can write the following assume statements for sets S_1 and S_2:

`inhale forperm` $[f]$ r `::` r `in` S_i
`inhale forall` r `: Ref ::` r `in` S_i `==> perm` $(r.f)$ $>$ `none`

where f is any field of the struct T_i. These `inhale` statements set the contents of S_i to be the references we currently have permission to and that were of type T_i in the original GRASShopper program. Using this improved encoding of such GRASShopper assume statements, our encoding of an exhale statement in GRASShopper that is translated back to Viper would now operate on properly initialized sets.

Conjunctions GRASShopper supports three types of conjunctions: `&*&`, `&+&`, and `&&`. Viper only supports the separating conjunction `&&`. Therefore we translate `&*&` to `&&` in Viper. Our translation currently also translates the GRASShopper conjunction `&&` to Viper's separating conjunction `&&`, which is correct if either the left or right hand side of the conjunction is a pure expression. If they are non-pure however GRASShopper expects

the permissions mentioned on both sides to be the same, which our translation does not reflect. The conjunction `&+&` is not supported by our translation, since it is not clear how to express the fact that the permissions mentioned on both sides of the conjunction can overlap in Viper. Note that `&+&` is never generated in our translation from Viper to GRASShopper, while `&&`, where both sides are non-pure, is generated by our translation of an `exhale` statement containing a predicate application (see section 3.9.2).

Functions Viper distinguishes between heap-independent domain functions and potentially heap-dependent functions outside of domains. GRASShopper does not make this distinction explicitly. In our translation, we translate all GRASShopper functions that have neither preconditions, postconditions, nor a body to domain functions. We create one domain for all the domain functions. All the functions that do not meet those requirements are translated to Viper functions outside of domains.

Predicates If we were to translate GRASShopper predicates to predicates in Viper, we would have to add appropriate `unfold` and `fold` statements to the translated code for every predicate application. Instead our translation assumes that all the predicates in the supplied GRASShopper program are not recursive. This assumption lets us replace all predicate applications in the translation by the predicate body itself.

Axioms In GRASShopper, axioms appear as top-level constructs, in Viper axioms need to be nested into domains. Axioms in Viper may mention domain functions from any domain, not just their own. In our translation we create one domain that contains the translations of all the axioms in the GRASShopper program. Note that we do not support the translation of axioms that mention GRASShopper functions which are translated to functions outside of domains.

Triggers We translate a matching trigger `@(matching` $t_{i,1}, \ldots, t_{i,n}$ `yields` e `)` to an e-matching trigger `{` $[\![t_{i,1}]\!], \ldots, [\![t_{i,n}]\!]$ `}` in Viper. This translation only approximates the behavior of triggers in GRASShopper since it ignores the expression e, in particular it ignores whether e has the form `known` (e') or not. Our translation does not support pattern triggers.

Set Comprehension We translate a set comprehension `{` $x : T$ `::` b `}` to a function application in Viper. First, we gather all subexpressions $e_1, \ldots, e_n$ of b that reference local variables, which are defined in the context where the set comprehension expression is located. This is the same technique we use in section 3.3, except that we do not restrict subexpressions to be of a specific type here. From the gathered subexpressions we construct parameters for the new Viper function $f_{setcomp}$. This function has return type `Set[` $[\![T]\!]$ `]` and postconditions that make sure the result of the function contains the same elements as the set specified by the set comprehension:

`ensures forall` $x : [\![T]\!]$ `::` b' `==>` x `in result`
`ensures forall` $x : [\![T]\!]$ `::` x `in result ==>` b',

where b' corresponds to the translation of b, but with the expressions $e_1, \ldots, e_n$ replaced by the corresponding parameter names. Any matching triggers that were attached to the set comprehension, we translate according to the previous paragraph and attach to both of the forall expressions in the postconditions. The set comprehension expression can now be translated to $f_{setcomp}(e_1, \ldots, e_n)$.

Uninterpreted Types We translate an uninterpreted type `type` T in GRASShopper to an empty Viper domain `domain` T `{}`.

Unsupported Features Our translation does not support reachability predicates such as `Btwn` and `Reach`. We also do not support GRASShopper's built-in array syntax.

5 Evaluation

Our tool Natrix implements the translation from Viper to GRASShopper described in section 3. In this section we evaluate Natrix, both in terms of how many test cases of the Viper test suite we can verify correctly, and how fast the tool is compared to the other Viper backends. We also discuss the general limitations of our translation from Viper to GRASShopper.

5.1 Limitations

In this section we discuss the limitations of our translation from Viper to GRASShopper and describe how the translation could be improved; both in terms of supporting more features of the Viper language, and enhancing existing encodings.

Unsupported Features Because of the differences between the Viper and GRASShopper languages, our tool only translates a subset of Viper to GRASShopper. The following list gives an overview of features which are not supported:

Magic Wands GRASShopper does not support magic wands and it does not seem possible at the moment to express them in GRASShopper.

Fractional Permissions GRASShopper does not support fractional permissions. Our translation is restricted to programs using only `write` and `none` permissions.

Permission Expressions Although GRASShopper does not natively support permission expressions, our translation could be improved to support `perm` and `forperm` expressions. Since we do not support fractional permissions, a permission amount could be modeled by `Bool`. The translation of `perm` $(x.f)$ would first obtain the current permissions *Perms* as described in section 3.8, and then check whether $map_f(x)$ is in $Perms(struct_f)$. Similarly, the translation of `forperm` $[f]\ r :: e(r)$ would check $[\![e(r)]\!]$ for every element r in $Perms(struct_f)$.

Sequences and Multisets are not supported by GRASShopper.

Set Cardinality is not supported by GRASShopper.

Unique Functions Such functions are not supported in GRASShopper.

Generic Domains As described in section 3.7, our translation only supports domains without type parameters. Generic domains could be supported by instanciating copies of the domains for all the relevant type parameters. A similar approach is already implemented in the symbolic execution backend of Viper.

Inhale-exhale Assertion in Functions and Predicates See section 3.9.3

Goto GRASShopper does not support `goto` statements.

Labels Labeled `old` expressions are not supported in GRASShopper.

Folding and Unfolding Predicates During our translation all folding and unfolding operations are dropped. This simplifies programs containing predicate applications, since in Viper we can only assert the body of a predicate once it is unfolded. However by dropping folding and unfolding operations we give up the ability to report any failures that may happen due to such an operation. In Viper, a `fold` statement can fail for two main reasons:

Either the predicate instance is in a state such that it cannot be folded, or the predicate body cannot be verified. To report the second kind of problem, we could translate `fold` statements to assert statements in GRASShopper in the future.

Quantified Permissions Our translation of quantified permissions relies on the triggering of forall expressions and set comprehensions (see section 3.12). The forall expressions have triggers attached as follows:

`forall` $[\![x : T]\!]$ `::` $[\![b]\!]$ `==>` $inv_e([\![e]\!])$ `==` $[\![x]\!]$
  `@(matching` $[\![e]\!]$ `yields known` $(inv_e([\![e]\!]))$ `)`
`forall` $w : R$ `::` $[\![b]\!][inv_e(w)/[\![x]\!]]$ `==>` $[\![e]\!][inv_e(w)/[\![x]\!]]$ `==` w
  `@(matching` $inv_e(w)$ `yields` $[\![e]\!][inv_e(w)/[\![x]\!]]$ `)`

Our translation does not specify triggers for the set comprehension at the moment:

`acc({` $r_f : struct_f$ `::` $[\![b]\!][inv_e(inv_f(r_f))/[\![x]\!]]$ `&&` $c_p[inv_e(inv_f(r_f))/[\![x]\!]]$ `})`

In some scenarios we can therefore not properly verify the program. Finding the specific triggers to properly verify quantified permissions remains subject to future work.

Recursive Predicates We translate predicates directly from Viper to GRASShopper, including any recursive predicates. Many GRASShopper programs containing recursive predicates do not terminate at the moment. However, the authors of GRASShopper are planning to implement an automatic translation of recursive predicates to predicates using reachability predicates instead.

Error Reporting In cases where we check loop invariants manually (see sections 3.9.3, 3.12.6, and 3.8), it can happen that we report both that the loop invariant was not established and also that it was not maintained, instead of just reporting the first error. To prevent this from happening, the tool would need to filter error messages.

```
0  method test()
   {
2    var i: Int := 1
     var j: Int := 2
4
     while (true)
6    {
       assert i == 1  // should succeed
8
       assert j == 2  // should fail
10
       j := j+1
12   }
   }
```

Viper

```
0  procedure test()
   {
2    var i: Int := 1;
     var j: Int := 2;
4
     while (true)
6    {
       pure assert i == 1;  // fails
       pure assume i == 1;
8      pure assert j == 2;  // fails
       pure assume j == 2;
10     j := j+1;
12   }
   }
```

GRASShopper

Translation 7: Unchanged variable inside a loop.

Checking Unchanged Variable Inside Loop Consider translation 7. The assertion on line 7 should succeed, but does not in the GRASShopper translation. In the other Viper backends this problem is solved by injecting so-called *free invariants* that guarantee that variables that are unaffected by the loop remain unchanged. In our case this free invariant would be `invariant i == 1`. However, finding the correct invariant is not

always as trivial, in particular if the variable is modified before the loop. Supporting those free invariants in our translation is subject to future work.

```
field f: Int

method m(x: Ref)
  requires acc(x.f)
{
  assert x != null  // should succeed

}
```

Viper

```
... // Field translation

procedure m(x: R)
  requires acc(map_f(x))
{
  pure assert (x != null);  // fails
  pure assume (x != null);
}
```

GRASShopper

Translation 8: Asserting non-null on a reference.

Field Encoding Our encoding of fields as discussed in section 3.6 has the issue that we never obtain permission to objects of type R. Since those objects directly represent Viper references, checking if a reference is non-null will always fail. See translation 8 for an example. We could improve the field encoding by translating `acc` $(x.f)$ to

`acc` $(map_f(x))$ `&*&` x `!=` `null` instead of `acc` $(map_f(x))$.

Using this improved encoding we could verify that a reference is non-null, whenever we have permission to one of its fields. However, if we are in a state where we have permission to none of the fields of a reference x, and we write an assertion `assert acc` $(x.f)$ our tool would now report both an error that we do not have permission to $x.f$ but also an error that x may be null.

Error Reasons Viper and GRASShopper both distinguish between a failure that happened because we don't have permission to some field, and because the assertion cannot be satisfied. In some cases however GRASShopper reports the wrong kind of error.

Predicate Flattening Due to a bug in GRASShopper predicates are not always flattened properly. This means assuming some predicate `p(x,y)`, does not guarantee that you can later assert a different predicate with the same body. This bug affects our translation, since we wrap Boolean operators in predicates, in places where GRASShopper does not allow any Boolean operators (see section 3.3). See translation 9 for an example of this issue.

Triggers Viper only generates default triggers if there are no user-defined triggers, while in GRASShopper default triggers are always generated, in addition to any user-defined triggers. Therefore our tool reports fewer errors than the other backends, when there is a user-defined trigger that does not trigger all the facts needed to prove some assertion.

Function Pre- and Postconditions GRASShopper only recently added support for specifying function pre- and postconditions. We directly translate Viper function pre- and postconditions to their corresponding GRASShopper representation. However the GRASShopper tool does not actually check function pre- and postconditions yet, so our tool fails to report violated function pre- and postconditions.

New Statement The GRASShopper `new` statement does not assume inequality of the new reference and references we currently don't have permission to. See translation 10 for an example.

```
function f(b: Bool): Bool


method test(x: Int, y: Int)
{
  inhale f(x != y);
  assert f(x != y); // should succeed

}
```

Viper

```
function f(b: Bool)
  returns (res: Bool)

procedure test(x: Int, y: Int)
{
  pure assume f(b_op(x,y));
  pure assert f(b_op1(x,y)); // fails
  pure assume f(b_op1(x,y));
}

predicate b_op(x: Int, y: Int)
{
  x != y
}

predicate b_op1(x: Int, y: Int)
{
  x != y
}
```

GRASShopper

Translation 9: Failing Boolean operator encoding.

```
field f: Int

method m3(x: Ref, y: Ref)
  requires acc(x.f)
{
  var z: Ref
  z := new(*)


  assert x != z  // should succeed

  assert y != z  // should succeed

}
```

Viper

```
... // Field translation

procedure m3(x: R, y: R)
  requires acc(map_f(x))
{
  var z: R;
  z := new R;
  pInhale(z);

  pure assert (x != z);  // succeeds
  pure assume (x != z);
  pure assert (y != z);  // fails
  pure assume (y != z);
}

procedure pInhale(new_var: R)
  ensures acc(map_f(new_var))
```

GRASShopper

Translation 10: New statement does not assume inequality for all references.

Well-Formed Assertions As discussed in section 3.2, Viper checks if expressions are well-formed in all positions where expressions can occur. GRASShopper only checks well-formedness for expressions that appear in assertions, assignments and procedure calls. Since our tool does not check well-formedness on its own, it fails to report well-formedness problems that GRASShopper does not find, i.e. not well-formed contracts, function bodies, and predicate bodies.

5.2 Viper Test Suite

We evaluated our new Viper backend Natrix using the test cases in the Viper test suite. We included the test cases from the `all` and `quantifiedpermissions` folders, but excluded the test cases from the `wands` folder, since those test cases are concerned with Viper magic wands, which our translation does not support. For the evaluation we ran the GRASShopper tool with the `z3` SMT solver, since `cvc4` rejects all test cases containing non-linear arithmetic, such as multiplication and division. The results are listed in table 1.

Success		132
Unsupported		315
	Wands	9
	Sequences	149
	Fractional permissions and permission expressions	119
	Unique functions	15
	Generic domains	8
	Inhale-exhale expression in function or predicate	4
	Multisets	3
	Set cardinality	3
	Goto statement	3
	Labels	2
Failures		58
	Well-formedness	19
	Folding and unfolding expressions are dropped	12
	Function pre- and postconditions not checked	13
	GRASShopper does not flatten predicates properly	7
	Assert false instead of insufficient permission	3
	Triggering of quantified permissions	2
	Recursive predicates in GRASShopper	2
	Testing non-null on reference fails	2
	New statement does not assume inequality	1
	Checking unchanged variable inside loop	1
	Report invariant not established and invariant not maintained	1
	GRASShopper always generates default triggers	1
	Insufficient permission instead of assert false	1
	Viper internal issues	6
Ignored		24
	Recursive predicates in GRASShoppers not terminating	10
	Viper internal issues	14

Table 1: Results in the Viper test suite

Of the 529 test cases in the suite 132 succeeded. 315 of the test cases contain at least one unsupported feature. Note that for each test case we counted only the first occurrence of an

unsupported feature. 58 test cases failed, which can be due to reporting unexpected errors, or failing to report an expected error. In each test case multiple errors can happen. For each type of error mentioned in the table we counted how many files it appears in.
Of the unsupported features, sequences appear to be the most frequent reason that a test case fails. Sequences could be supported by our translation by adding axioms that describe the behavior of sequences. Those axioms would be very similar to the axioms the other Viper backends use. But sequences are not the only feature that could be supported by adding appropriate axioms: multisets, set cardinality, and unique functions also fall into this category. As described in section 5.1, support for the permission expressions `perm` and `forperm` could also be added to the translation.
The number of test cases that fail because folding and unfolding operations are dropped could be reduced as discussed previously, but there will always be such errors, as we drop those operations is by design. The field encoding could be improved, such that testing whether a reference is non-null verifies properly. Most of the other failing test cases are due to bugs in the GRASShopper tool, or features that GRASShopper does not support yet. Some of those issues could be handled by our translation, such as the well-formedness checks and the checking of function pre- and postconditions. But once the GRASShopper tool supports those features itself, this work would be redundant. The problems arising because of the differences between the triggers in Viper and GRASShopper need to be analyzed more deeply.

5.3 Performance

We measured the performance of Natrix and both existing Viper backends on the setup shown in table 3. We measured timings for Natrix, with both `z3` and `cvc4` as the backend SMT solvers for GRASShopper[8]. We ran the performance tests on the test cases described in section 5.2, with a timeout of 120 seconds and 5 repetitions. For the comparison we only consider test cases, in which all configurations finished before the timeout, and did not return an error. The results of the performance test can be found in table 2. We can see that our new backend is not much slower on average than the backend of Viper using symbolic execution (SE), but can be as slow or even slower than the backend using verification condition generation (VCG). Using both SMT solvers in parallel seems to improve the performance of GRASShopper and therefore Natrix slightly.

	Natrix `z3`	Natrix `cvc4`	Natrix `z3+cvc4`	Viper SE	Viper VCG
Average Runtime	1.80s	1.84s	1.77s	1.76s	3.91s
Maximum Runtime	6.59s	8.81s	6.94	2.79s	6.83s

Table 2: Results of performance tests

[8]When using `cvc4`, we ran GRASShopper with `-smtsolver cvc4mf`.

CPU	Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz
Memory	16GB
OS	Ubuntu 14.04.5 LTS
JVM	Java(TM) SE Runtime Environment (build 1.8.0_101-b13)
GRASShopper	commit `122172e8b7f0504977875548d584018be2bd5904`
`z3`	version 4.4.0
`cvc4`	version 1.5-prerelease
Silver	changeset `2202:7c482b3066e7`
Silicon	changeset `1467:53a4f89e0b11`
Carbon	changeset `839:63e01a20b4f5`
Natrix	commit `ea6e00843fd914ec5751bb849e9dc208e2b2c76d`

Table 3: Setup used for testing.

6 Conclusion and Future Work

We have presented a translation from a substantial subset of the Viper language to the GRASShopper input language. We have implemented this translation as a new backend verifier for Viper. Possible improvements of this translation include adding support for generic domains and sequences. We also have laid the foundations of a translation from GRASShopper back to Viper. Main improvements on this translation include support for reachability predicates, encoding of precise permission checking and inside expressions.
A translation of recursive predicates to non-recursive predicates using GRASS reachability predicates instead would be interesting from both the GRASShopper and the Viper perspective. Using our translation from Viper to GRASShopper and the translation of the recursive predicates, we would be able to better understand the connections between recursive and point-wise definitions of data structures.

Acknowledgements We thank Alexander J. Summers for his detailed explanations of Viper, helpful discussions on the translation, and detailed feedback on earlier versions of this report. We are grateful to Thomas Wies, for answering all of our questions on GRASShopper and investing substantial time and effort making modifications to GRASShopper based on our feedback. We thank Steven Köppel for his feedback on the final draft of this report.

Appendices

A Breaking up an Assertion into a Sequence of Statements

As described in sections 3.9.1, 3.9.2, and 3.12.1, it is often nessecary to break up an assertion into subexpressions that can be handled sequentially. Because pure and non-pure subexpressions often have to be handled differently, we want to distinguish between pure subexpressions, predicate accessibility predicates, field accessibility predicates, and quantified permissions. In this appendix, we describe how to break up assertions into subexpressions. These subexpressions are then translated into sequences of statements, according to the descriptions in the sections mentioned above.
We break up the assertion recursively. Since pure expressions are not broken up further, we only need to recurse when we encounter a subexpression that can join two assertions. There are only three of those, namely conjunction, implication, and conditional assertion. We translate those assertions into the corresponding control flow statements, while recursing on their subexpressions.
This approach is illustrated in algorithm 2.

Algorithm 2 Breaking up a Viper assertion into statements

```
1:  function BREAKUP(e)
2:      switch e do
3:          case e is pure
4:              return list of statements corresponding pure expression e
5:          case acc (p(e_1, ..., e_n))
6:              return list of statements corresponding to predicate access e
7:          case acc (e'.f)
8:              return list of statements corresponding to field access e
9:          case forall x : T :: b(x) ==> acc (e'(x).f, p(x))
10:             return list of statements corresponding to quantified permission expression e
11:         case e_left && e_right
12:             return BREAKUP(e_left) ; BREAKUP(e_right)
13:         case e_left ==> e_right
14:             return if ([[e_left]]) { BREAKUP(e_right) }
15:         case e_cond ? e_then : e_else
16:             return if ([[e_cond]]) { BREAKUP(e_then) } else { BREAKUP(e_else) }
```

References

[1] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[2] R. Piskac, T. Wies, and D. Zufferey. *GRASShopper: Complete heap verification with mixed specifications*, volume 8413 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 124–139. Springer Verlag, 2014.

[3] R. Piskac, T. Wies, and D. Zufferey. *Automating separation logic with trees and data*, volume 8559 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 711–728. Springer Verlag, 2014.

[4] R. Piskac, T. Wies, and D. Zufferey. *Automating separation logic using SMT*, volume 8044 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 773–789. 2013.

[5] P. Müller, M. Schwerhoff, and A. J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In *Computer Aided Verification (CAV)*, LNCS. Springer-Verlag, 2016. To appear.

[6] L. de Moura and N. Bjørner. *Efficient E-Matching for SMT Solvers*, pages 183–198. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[7] A. J. Summers and S. Drossopoulou. A formal semantics for isorecursive and equirecursive state abstractions. In *European Conference on Object-Oriented Programming*, pages 129–153. Springer, 2013.