Advancing Non-Standard Permission Utilisation in Program Verification

Bachelor's thesis

Tobias Brodmann
Supervisors: Prof. Dr. Peter Müller, Dr. Malte Schwerhoff

05.03.2018

1 Introduction

The Viper project [4], which is developed at ETH Zurich, provides an intermediate verification language and verification tools for encoding a wide variety of concurrent verification problems. A key feature of Viper's approach to verification is the notion of *permissions* to control access to and formally reason about resources: in Viper, each heap location is such a resource (e.g. `this.val`); as are *recursive predicates* and *magic wands*, which can be used to control access to and enforce orderly modification of complex data structures [1].
Viper includes two verifiers for automated reasoning about verification problems involving such resources: Silicon [2], which is based on a technique called symbolic execution, and Carbon, which generates verification conditions.
Ultimately, both verifiers generate formulas which are discharged by an SMT solver.

Viper's support for standard utilisation of permissions, i.e. the checking and transferring of permissions between verification contexts, is well-developed; among other things, because the theory behind this kind of utilisation has been extensively researched in the last two decades.
As more and more complicated and complex programs and systems need to be verified, this standard use of permissions is no longer enough and new features are needed.
For these new problems, Viper additionally also supports non-standard utilisation of permissions, such as **`forperm`** expressions which serve as a quantifier over all objects for which a specified permission is currently held.
These are unique to Viper and have been shown to be immensely useful in recent projects such as [3].
However, they are not yet fully compatible with already existing Viper features. This can lead to unexpected errors for users, because some combinations of features are allowed while others are not.

An example of this problem is *permission introspection*, the querying of permissions and acting depending on their availability. Consider the following code snippet:

```
if (perm(P --* Q) == write) {
        exhale P --* Q
} else {
        exhale R
}
```

Listing 1.1: Permission introspection

In this example, we want to *exhale* a permission, i.e. give away the access permission to a resource, but which permission depends on the currently held permission. This is useful because one permission might be more valuable or more important to us, and therefore we want to control which permission is exhaled. The `--*` operator in this example denotes the *separating implication* or *magic wand* in separation logic. While permission introspection in general is possible in Viper, this particular combination with a magic wand is not, which leads to confusing issues for users.

Another example of such a problem is affecting the verification state by making assumptions about the currently held permissions. For example:

```
//acc(y.f)
assume acc(x.f)
//acc(y.f), x = y
```

Listing 1.2: **assume** with non-pure assertion

Here, a client holds the permission to `y.f` before the **assume** statement (denoted by **acc**`(y.f)` in the preceding comment). Since that is the only permission held, we learn that `x = y`, because we assume that permission to `x.f` is already being held. Assuming that permissions are already held is not yet supported, in contrast to assuming pure (non-permission) assertions.

2 Core Goals

- Fully support *permission introspection* with **perm** and **forperm**; in particular, fully support predicates, quantified permissions and magic wands. Currently permission introspection can be used with the following Viper features:

P	**perm**	**forperm**
`e.f`	✓	✓
`P(e1,e2,...)`	✓	✗
`A --* B`	✗	✗

Table 2.1: Current support for permission introspection

  In this table, **P** denotes the resource that is used for the introspection.

- Develop a technique to use non-pure assertions with **assume** statements. Similar to permission introspection, non-pure assertions can also already be used in many cases, but not in the particular combination with **assume**:

Support for assertions	
exhale P	✓
inhale P	✓
assert P	✓
assume P	✗ (if not pure)

Table 2.2: Support for non-pure assertions

Note that, in listing 1.2, it would be possible to simply replace the non-pure assertion **acc**(x.f) by the pure assertion **perm**(x.f) == **write**. This would lead to exactly the same result. But consider the following example:

```
//acc(a.f), acc(b.f), a != b
assume acc(x.f) && acc(y.f)
//acc(a.f), acc(b.f), a != b, (x = a && y = b) || (x = b
    && y = a)
```

Listing 2.1: **assume** with *separating conjunction*

If we used the same replacement strategy as in the previous example, this would result in the following:

```
//acc(a.f), acc(b.f), a != b
assume perm(x.f) == write && perm(y.f) == write
//acc(a.f), acc(b.f), a != b, (x = a && y = b) || (x = b
    && y = a) || (x = y = a) || (x = y = b)
```

Listing 2.2: **assume** with *boolean conjunction*

However, as can be seen in the comments denoting the verification state, the effects are not the same even though they look very similar. The reason for that is the overloading of the `&&` operator. While usually it denotes a standard boolean conjunction, when used with accessibility predicates it denotes *separating conjunction* (denoted $*$ in separation logic). Because of this, in the first example `x` and `y` are not allowed to be *aliases*, i.e. references to the same memory location, whereas in the second example, they could be aliases. Therefore, it is not straight-forward to replace accessibility predicates by **perm** expressions and special support is needed.

- Support the use of permissions and heap locations as triggers in quantified expressions: Viper supports the use of *quantified expressions*, which are passed on to the underlying SMT solver. To decide where to instantiate the quantified expression, SMT solvers use *triggers*. For example:

```
forall i: Int, j: Int :: {f(i), g(j)} f(i) && i < j ==>
    g(j)
```

Listing 2.3: Quantified expression

In this example, `f(i)` and `g(j)` are used as the triggers. Currently, Viper does not allow the use of permissions and heap locations as triggers, but supporting such triggers is required in order to properly handle quantifiers such as:

```
assume forall x: Ref :: perm(x.f) > none ==> x.f != 0
```

Listing 2.4: Quantifier with heap-based trigger

Here, we want to assume that for all `x` to whose field `f` we have access to, `x.f` is not 0. The only reasonable triggers for this quantifier are `x.f` and **`perm`**`(x.f)`, both of which are currently not supported.
In general, all Viper resources (fields, predicates and magic wands) should be usable as triggers.

3 Extension Goals

- Design and implement an extension to the Viper language to allow "pattern matching" with **`forperm`**. This would enable more flexible ways of specifying the required resources, such as predicates. For example the syntax could look like **`forperm`** `P(?x, 3) :: e(x)`, where `?x` specifies the variable to be matched, i.e. it would match every predicate instance for which the second argument is 3. A similar syntax could also be created for magic wands. An even more flexible extension would be to allow binding variables to arbitrary expressions such as `P(?x + 1, 3)`.
- Enable Silicon to recover from failures. In contrast to e.g. most compilers, which can recover from type checking failures, Silicon cannot recover from a failing assertion. This is annoying in practice because it means that errors resulting from failing assertions need to be manually handled one at a time. However, support for assuming arbitrary assertions A should enable Silicon to turn any failing assertion into an assumption and thereby to continue the verification.
- Investigate the potential usefulness of extending Viper with support for marking resources, in particular fields, as non-exclusive. Such a feature might simplify the encoding of programs with fine-grained concurrency, but will raise questions such as how to frame assumptions about such resources.

References

[1] P. Müller, M. Schwerhoff, and A. J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by B. Jobstmann and K. R. M. Leino. Vol. 9583. LNCS. Springer-Verlag, 2016, pp. 41–62.

[2] M. Schwerhoff. "Advancing Automated, Permission-Based Program Verification Using Symbolic Execution". PhD thesis. ETH Zurich, 2016.

[3] A. J. Summers and P. Müller. "Automating Deductive Verification for Weak-Memory Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. LNCS. To appear. Springer-Verlag, 2018.

[4] Viper Project. URL: `viper.ethz.ch`.