# Abstract Read Permission Support for an Automatic Python Verifier

Bachelor Thesis Project Description

Benjamin Schmid
Supervised by Vytautas Astrauskas, Marco Eilers, Prof. Dr. Peter Müller
Department of Computer Science
ETH Zürich
Zürich, Switzerland

## I. Abstract Read Permissions

In program verification there are a lot of cases where one needs to reason about access to memory locations. Two such cases are data races and the framing problem.

Data races occur in a concurrent program if several threads access the same memory location in an unsynchronized way and at least one of them is a write operation. The program behaviour depends on the exact order of memory accesses and will be different for different runs of the program. This results in hard to find bugs.

Framing is the problem of what assumptions can be preserved across a method call. When calling a method the caller does not know which values are not modified by a method call. Therefore, it has to drop all assumptions about all global variables and fields of objects. The only thing the caller can assume after the call is the postcondition of the method. For example a method with `x.f = x.g + 1 && x.g == old(x.g)` as a postcondition only makes a statement about the value of `x.f` and `x.g`. The caller does not have any information about any other field (e.g. `x.h`) and whether it was modified by the method.

Both problems can be solved by using verification based on permissions. Methods specify in their pre- and postconditions the needed permissions to fields. If they lack the read or write permission to a field they are not allowed to read or write this field. The verifier then knows which memory locations might have been changed and which still hold the same value. If a thread holds a partial permission it can be sure that no other thread can change this value.

For each heap location there exists in total at most a permission amount of 1. The permission for a location can be split, transferred between methods and recombined again. To write to a heap location a method needs the full permission (i.e. a permission amount of 1). To read it needs any positive amount of the permission. As soon as no permission is left within a method all assumptions about the location will be dropped. This allows to preserve assumptions about heap state across method calls and to prove the absence of data races in concurrent programs.

We will use the example in Listing 2 from [1] to illustrate the problem we will be working with in this project. In the example, we have a class `Expr` representing a node in an arithmetic expression and a method `eval(s: State)` to evaluate the expression in the given state (Listing 1).

```
class Expr {
  ...
  method eval(s: State)
    requires acc(s.map, π) && s.map != null
    ensures acc(s.map, π)
  ...
}
```

Listing 1: Class Expr

This method requires read permission to the field `s.map`, which is denoted by `acc(s.map, π)`. The precondition transfers π permissions from the caller to the callee while the postcondition will transfer it back to the caller. For simplicity we assume the fields left and right are immutable and thus no permission is needed to access them.

```
class Add extends Expr {
  var left, right: Expr

  method eval(s: State)
    requires acc(s.map, π) && s.map != null
    ensures acc(s.map, π)
  {
    leftVal := call left.eval(s)
    rightVal := call right.eval(s)
    return leftVal + rightVal
  }
}
```

Listing 2: Example illustrating problem: Class Add

The problem is how to choose π. Using concrete fractions (e.g. `π = 1/2`) does not work. The first recursive call would give up all permissions the method held and the caller can not prove `s.map != null` any more as knowledge about `s.map` was havocked. Further it is not possible to call `eval` in a context where the caller only holds a permission of for example `1/4`. As the callee only needs to be able to read the location and the caller has a positive amount this should be possible. But due to the concrete fractions it is not. Moreover the concrete value of `π` does not hold any additional information and could be any other positive amount less than 1 as we just want to denote any read permission.

Another approach is to use counting permissions [2]. The permission is split into infinitesimally small, indivisible units. Having such a unit grants read access. This allows to call the method from any context which has the necessary read permissions and the user does not need to choose a concrete fraction. But the first call will still use up all permission held and thus will not allow us to verify the example.

A third approach uses ghost parameters. This is an additional parameter being passed to a method (see listing 3) which is only used for verification and does not exist at runtime. The programmer would need to explicitly choose what permission amount to give to the callee (e.g. `call left.eval(s, π/4)`). But how the permission is split does again not give any additional information. While this allows to verify our example it is cumbersome for the programmer and should not be necessary.

```
method eval(s: State, ghost π: rational)
  requires 0 < π && π <= 1 && acc(s.map, π)
  ensures acc(s.map, π)
```

Listing 3: Ghost parameter

In [1] abstract read permissions (abbreviated as ARP) are presented. The concrete values in the code are replaced by `rd` (see listing 4). When verifying a method the `rd` qualifier within this method always represents the same value. When the method is called the qualifier can be instantiated with a suitable value but it might be different for each invocation. This allows to specify that the postcondition will return the same amount of permission the method got in the precondition. ARPs solve the drawbacks mentioned above.

```
method eval(s: State)
  requires acc(s.map, rd) && s.map != null
  ensures acc(s.map, rd)
```

Listing 4: Abstract read permissions

For the encoding of ARP it is not needed to choose a concrete value for `rd` at any point. The verifier can constrain its value according to pre- and postconditions [3] [4].

## II. Nagini

Nagini is a verifier for statically typed Python programs currently being developed at ETH. It uses the Viper verification framework [5], a verification infrastructure for permission-based reasoning. Viper provides an intermediate language which can be used to implement front ends for different languages (e.g. Nagini for Python). The Viper language is human readable and can be written manually. This allows for fast prototyping of new verification approaches. Viper and Nagini do not support abstract read permissions at the moment. The goal of this project is to implement ARP in an extension to the Viper language. Nagini will then be updated to use this extension and thus have support for ARP.

## III. Core Goals

1) Design an extension to the Viper language which supports ARP. The extension has to support the `rd` qualifier in standard `acc(x.f, rd)` access predicates, counting permissions `rd(n)`, permission expressions containing those terms (e.g. `acc(x.f, 1/2 + 3*rd - rd(2))` and wildcard permissions (`acc(x.f, rd*)`, `rd*` denotes a different value in each occurrence). It does not need to support ARP in quantified expressions (see extension goals). Designing the extension consists of specifying syntax and describing how the added features can be modeled in the Viper language.
2) Specify new syntax for Nagini to make it possible and easy to use ARP in Python. The new syntax should enable use of all features added in the extended Viper language.
3) Develop a translation from the extended Viper language to the Viper language. The translation gets as input a source file using features from the extended Viper language and produces as output a semantically equivalent source file in the Viper language. Further it allows to map errors occurring in the translated program back to the input source code. In his Master's thesis [6] Simon Fritsche implemented a framework which simplifies this kind of translations as well as the mapping of error messages back to the original source code. We aim for a composable implementation of the translation such that several extensions to Viper can be combined. Depending on the design of the extension and the details of the mentioned framework this might not be achievable. Furthermore, the implementation should not duplicate code of the Viper language implementation. The performance of this translation and the verification of the translated program has to be monitored. Found performance issues should be fixed if possible.
4) Make ARP available in Nagini by designing and implementing a mapping from Nagini to the extended Viper language. For this the syntax specified in 2 will be used. One should be able to model the first example from [3] and all ARP tests from the Chalice2Viper[1] test suite. Errors occurring in the verification of the generated Viper code can

[1]https://bitbucket.org/viperproject/chalice2silver

be mapped back to the originating code snippet in Python.

5) Evaluate the implementation using typical examples from the implementation of SCION [7] and from the Chalice test suite. This comprises testing speed and completeness of the implementation. Moreover, the performance difference between using concrete fractions and using ARP for the same example will be evaluated.

## IV. Extension Goals

- Enhance the extended Viper language to support quantified abstract read permissions (that is for example `forall x:Ref :: x in S ==> acc(x.first, rd)`)
- Make quantified abstract read permissions available in Nagini by further extending the syntax designed in core goal 2 and enhancing the implementation written during core goal 4.
- Integrate ARPs for permission inference in Sample[2], a static analyzer being developed at ETH.

## V. Schedule

| | |
|---|---|
| 18.09.2017 | Start |
| 05.10.2017 | Initial Presentation |
| 20.10.2017 | Design of extension to Viper language |
| 27.10.2017 | Specify new syntax for Nagini |
| 24.11.2017 | Translation to extended language |
| 15.12.2017 | Update Nagini |
| 05.01.2018 | Evaluate implementation |
| 22.01.2018 | Start extension goals |
| 29.01.2018 | Start writing report |
| 18.03.2018 | Deadline |

## References

[1] S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers, "Abstract read permissions: Fractional permissions without the fractions," in *International Workshop on Verification, Model Checking, and Abstract Interpretation.* Springer, 2013, pp. 315–334.

[2] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," in *ACM SIGPLAN Notices*, vol. 40, no. 1. ACM, 2005, pp. 259–270.

[3] J. T. Boyland, P. Müller, M. Schwerhoff, and A. J. Summers, "Constraint semantics for abstract read permissions," in *Proceedings of 16th Workshop on Formal Techniques for Java-like Programs.* ACM, 2014, pp. 1–6.

[4] V. Astrauskas, "Encoding of chalice permissions in viper," unpublished, 2017.

[5] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62.

[6] S. Fritsche, "A framework for bidirectional program transformations," Master's Thesis, ETH Zurich, 2017.

[7] D. Barrera, L. Chuat, A. Perrig, R. M. Reischuk, and P. Szalachowski, "The scion internet architecture," *Communications of the ACM*, vol. 60, no. 6, pp. 56–65, 2017.

[2]https://bitbucket.org/viperproject/sample