Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract Read Permission Support for an Automatic Python Verifier

Bachelor Thesis

Benjamin Schmid

Friday 16th March, 2018

Advisors: Prof. Dr. Peter Müller, Vytautas Astrauskas, Marco Eilers
Department of Computer Science, ETH Zürich

Abstract

In program verification there are a lot of cases, like for example the verification of absence of data races, where one needs to reason about access to memory locations. By using verification based on implicit dynamic frames [14], a logic based on access permissions, one can reason about such kind of problems.

In concurrent programs often multiple readers should be able to access a location at the same time. For this, it is necessary to split the permission such that each thread has some permission. One way of doing this, is to use rational numbers in the range $[0, 1]$ to model permissions, where 1 is a full permission granting write access and > 0 permission grants read permission.

When choosing values for access permissions it is sometimes difficult to choose a suitable permission value to represent a read permission. Abstract read permissions [4, 10] allow specifying some read permission to a resource (e.g., a memory location) without choosing a concrete permission amount. This value is suitably constrained by the verifier such that it is positive but less than the currently held permission amount. Thus, for example in a method call, the caller does not transfer all held permission and does not need to havoc any information.

The Viper verification framework [12] supports reasoning based on implicit dynamic frames. In this project we implemented abstract read permissions for Viper. We describe a new encoding based on new insights published by Boyland et al. [4], which is more powerful than existing approaches and allows to verify more complex examples. Support for abstract read permissions was added to Nagini [7], a front-end for Viper to verify statically-typed Python programs.

Acknowledgements

I would like to thank my supervisors Vytautas Astrauskas and Marco Eilers for their immense support during the project. Their feedback and inputs were very valuable and our weekly meetings provided many interesting and enlightening discussions and helped me to stay motivated. I would also like to thank Prof. Dr. Peter Müller for the opportunity to work on this interesting project.

Contents

Chapter 1

Introduction

Software is becoming increasingly parallel. Concurrent programs are difficult to get right and therefore, one wants to be able to verify their correctness. Besides functional correctness, this also includes the absence of data races. This requires us to be able to reason about heap access. One approach to this is to use deductive verification based on a permission logic. In deductive verification, programs are verified according to a specification given by the user. To make it scalable the program is being verified modular during verification: each method is verified independently and only method contracts are considered for method calls.

One approach to modular verification with the ability to reason about memory access are implicit dynamic frames [11, 14], a permission logic using fractional access permissions to resources such as memory locations. Each memory location has a total of one permission associated and this permission can be split between methods. A thread can write to a memory location if and only if it holds the full permission to that memory location. To read the location it needs to hold some positive permission amount. If a thread holds some positive permission to a memory location, then no other thread can have full permission to that memory location. Thus, the location cannot be written by any other thread. On the other hand, if no permission is held the location can be changed by another activation record at any moment and the thread cannot retain any information about the content of the memory location. In this way, implicit dynamic frames allow reasoning about concurrent programs and enables verification of absence of issues such as data races.

It is sometimes difficult to choose suitable permission amounts if a read permission has to be specified. We consider the example in Listing 1 and 2 to illustrate the problem (taken from the 2013 VMCAI paper by Heule et al. [10]). We have a class `Expr` (Listing 1) representing a node in an arithmetic expression. For simplicity we assume the fields `left` and `right` to be immutable and thus no permission is needed to access them.

The method `eval` evaluates the node in the given state. It needs to be able to read the field `s.map` which models a mapping from variables to values. The method will not change it and thus only some positive permission amount is needed. The method in the class `Add` has the same method contract as the method it overrides.

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

Listing 1: Class `Expr` represents a node in an arithmetic expression.

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

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

Listing 2: Example illustrating problem of choosing suitable read permission: Class `Add`. Method `eval` has to be able to read `s.map` but it is not clear how to choose the permission amount.

The problem is now how to choose `π`, i.e., the amount of permission that should be transferred from the caller to the callee.

Choosing a concrete value for `π` (e.g., `π = 1/2`) does not work. While it will give the callee permission to read `s.map` it is not possible for the caller to retain the assumption `s.map != null` across the method call. The reason is that the caller itself only has a permission of `1/2` and thus it has to give up all permission. All information about `s.map` is therefore havocked. Hence, the second call to `right.eval(s)` will fail because the assumption `s.map != null` had to be dropped. Another problem is that it would not be possible to call `eval` from a context where the caller holds some permission but less than `1/2`.

Intuitively this should be possible as `1/2` in this case just represents a read permission and the chosen fraction is rather arbitrary.

Another approach are counting permissions [3]. The permission is split into infinitesimally small, indivisible units. Holding at least one unit allows to read the associated location. While this will allow to call `eval` from any context (by specifying `π = rd(1)`, i.e., a single unit) it will still not work due to the first problem of giving up all permission.

What one really wants to specify is that the location has to be readable. As long as the assertion is well-formed a suitable, positive permission amount exists. Just specifying some read permission also removes the need to choose an arbitrary value for the permission which has no further meaning (besides granting read access). In the 2013 VMCAI paper, Heule et al. [10] present abstract read permissions (abbreviated as ARP). The paper introduces an `rd` qualifier resulting in the specification shown in Listing 3. `rd` specifies that a read permission is needed. Each `rd` qualifier will be replaced by a suitable value which is positive and strictly less than the amount held by the caller. However, it is not necessary for the verifier to choose a specific value. It suffices to automatically constrain it such that it is positive and small enough to leave the caller with some permission [1, 4]. An interesting feature of `rd` is that in different places it can refer to different permission amounts. For example, in two subsequent method calls it does not refer to the same amount. However, a common pattern is that the callee temporarily needs read access to some structure. This pattern is supported by making `rd` refer to the same value in a called method's specification during the call, which allows the callee in the postcondition to give back all permission which was transferred by the precondition. This is important for the caller to be able to get back the full permission.

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

Listing 3: Contract for example in Listing 2 using `rd`, an abstract read permission.

Abstract read permissions were first implemented in Chalice [11]. In this project we describe a new encoding for abstract read permissions which allows to verify more usages and does not rely on ad-hoc solutions. The new encoding is based on a theoretical generalization of ARP which provides a new, more general sufficient condition for a sound encoding [1, 4].

The new encoding was implemented as an extension to the Viper verification framework [12] which is being developed at ETH Zurich. Viper consists of an intermediate language and back-ends used to verify this intermediate language. It allows fast prototyping of new verification approaches and can be used for front-ends in different languages (such as VerCors [2] for Java or Nagini [7] for Python). The implemented extension provides an extended Viper language which is translated into normal Viper and thus can be verified using the existing infrastructure.

The added features were made available in Nagini [7], a front-end for Viper to verify statically-typed Python programs.

We evaluated the new encoding by implementing examples from the Chalice2Viper [6] test suite and some examples which were not possible in Chalice. Furthermore, the performance of the encoding was evaluated and the effect of using abstract read permission on the verification time was measured.

Outline. Chapter 2 introduces necessary background for the following parts. The new encoding is presented in Chapter 3. In Chapter 4 we describe the implementation of the translation from the extended Viper language to normal Viper. Finally, in Chapter 5, the new encoding is evaluated in terms of performance and new examples which can now be verified.

1.1 Project Goals

This project had the following goals:

1. Design an extension to the Viper language to support abstract read permissions. This includes `rd` qualifier in access predicates, counting permissions and wildcard permissions.
2. Specify new syntax for Nagini to use abstract read permissions.
3. Develop a translation from the extended Viper language to the Viper language. It has to be possible to map errors back to the original program. The translation should be composable.
4. Implement support for abstract read permissions in Nagini. It should be possible to model the first example from the 2014 FTfJP paper by Boyland et al. [4].
5. Enhance the extended Viper language to support abstract read permissions in some instances of quantified expressions.
6. Make quantified expressions available in Nagini.
7. Evaluate the implementation using examples from the Chalice2Viper [6] test suite.

Chapter 2

Background

This chapter introduces necessary background on which the next parts build upon. In Section 2.1 we give a short introduction to permission-based verification. Section 2.2 describes the structure of Viper programs. Nagini is shortly introduced in Section 2.3. Finally, Section 2.4 describes the existing ARP encoding in Chalice.

2.1 Permission Based Verification

We will first look at two examples of problems which can be solved with the verification approach described below: prevention of data races as part of verifying concurrent problems as seen in Chapter 1 and the framing problem. A data race is defined as some memory location being accessed concurrently by several threads in an unsynchronized way where at least one of the accesses is a write. If one of them is a write the behavior of the whole program depends on the exact order of memory accesses and the outcome is in many programming languages not defined.

Framing is an important problem in modular verification, which is needed to be able to scale to large programs. In modular verification only pre- and post-conditions of called methods are considered for verification and the method body is ignored. All methods are verified separately in isolation from each other. Framing is the problem of which assumptions can be kept across method calls. Without any additional information the caller does not know which heap locations may have been changed by the callee. Specifying all unchanged memory locations is not always possible and in cases where it would be possible it does not scale to larger programs. Furthermore, it would not be modular to always have to mention all memory locations.

Verification based on implicit dynamic frames [11, 14], a permission-based approach, enables framing and verification of concurrent programs, including verifying absence of data races. We consider implicit dynamic frames with fractional permissions, where permissions are modeled as rational numbers in the range $[0, 1]$. Each memory location has one full permission associated with it. This permission can be split infinitely and it can be held by methods, loops, locks and monitors. To be able to read a memory location, a method invocation or loop iteration has to hold some positive amount of permission. To write to a memory location the full permission (a permission amount of 1) is needed.

Required permissions are specified in pre- and postconditions for methods and in loop invariants for loops. Upon calling another method or entering a loop the requested permission amount is transferred (after checking that enough permission is held by the caller/surrounding verification scope).

If the full permission is held for a specific location one can be sure that no other method holds any permission to this location and it can be modified without the danger of data races. Further, if some positive amount is held the value cannot be changed by a called method or another thread as it is not possible for the other method/thread to hold a full permission. This allows framing because information can be kept across a method call without the need to explicitly specify all unchanged locations. On the other hand, if the caller holds no permission, the callee may have a full permission and thus the caller has to havoc all knowledge about this memory location.

Listing 4 shows an example how permissions can be used. `acc(x.f)` is used to represent write permission to the field `f` of object `x`. `acc(x.f, 1/2)` is the same but only for half the permission which only grants read access.

2.2 Viper

The Viper verification framework [12] is a verification infrastructure based on implicit dynamic frames. It is being developed at ETH Zurich. Viper provides an intermediate language which can be used to implement front-ends for different languages (e.g., Nagini for Python [7], VerCors [2] for Java). The intermediate language used by Viper (subsequently called Viper language) is human readable and can thus be written manually. Viper provides two back-ends to verify programs. One is based on symbolic execution, and the other is based on verification condition generation. This section introduces the main features of Viper which are relevant to this project. For a more detailed discussion of Viper see the paper introducing Viper [12].

A Viper program consists of some fields, predicates, functions, methods and domains. Before we can describe these parts it is necessary to explain how permissions to resources are managed.

```
method calleeRd(x: Ref)
  requires acc(x.f, 1/2)
  ensures acc(x.f, 1/2)
{
  x.f := 4          // fails, not enough permission
}

method calleeWrite(x: Ref)
  requires acc(x.f)
  ensures acc(x.f)
{
  x.f := 42         // succeeds
}

method client(x: Ref)
  requires acc(x.f)
  ensures acc(x.f)
{
  x.f := 42         // we have full permission
  calleeRd(x)
  assert x.f == 42 // succeeds, body of calleeRd is ignored
  calleeWrite(x)   // only pre- and postconditions are considered
  assert x.f == 42 // fails, x.f might have been changed
}
```

Listing 4: Example of permissions. Comments show which statements succeed and which fail.

To manage permissions to resources, assertions can be exhaled and inhaled. The first assumes the stated assertion and adds all permissions mentioned to the current state. Exhaling does the opposite: it asserts the stated assertion and removes permissions.

Access predicates are used to specify a permission amount to a certain location. A required permission is specified using `acc(x.f, perm_amount)` to access a field. As we will see there are also permissions to predicates, which are specified by `acc(pred(x, y, z), perm_amount)` (where x, y and z are parameters of the predicate called `pred`). `perm_amount` can be any value between 0 (written as `none`, no access at all) and 1 (written as `write`, granting write permission). `acc(x.f, write)` can be written as `acc(x.f)`. If multiple access predicates to the same location are present the permission values will be summed up (e.g., `acc(x.f, 1/2) && acc(x.f, 1/2)` will grant write access). When referring to location accesses we mean both predicates and fields.

It is important to differentiate between expressions and assertions. An assertion is a boolean expression, but can in contrast to other expressions contain access predicates and can be used in specifications (e.g., `assert`, `inhale` and `exhale` statements). An assertion without access predicates is called a pure assertion.

A field represents a typed memory location on the heap. Every reference has all fields declared in the program and there is no notion of a class. A field access is always of the form `x.f` for a reference `x` and a field `f`. The reference can be any reference typed expression, for example a function result (`get_from_array(array, 0).f`) or another field access (`x.r.f`).

Predicates can be used to encapsulate assertions. They can have parameters which can be used in its body. Permissions cannot only be held to a certain memory location but also to a predicate. Two predicate instances of the same predicate are the same if and only if they have the same argument values. This is also the case if the argument which differs is not mentioned in the body at all (see Listing 5).

If a method holds permissions to a predicate the latter can be exchanged for its body. This is called `unfold` in Viper. The reverse is also possible: with `fold` a predicate can be recreated if the assertion in its body holds in the current state. When exchanging a predicate for its body the permission to the predicate is exhaled, that is it is asserted and the permissions are removed, and the body is inhaled, i.e., it is assumed and the permissions are added. The analogue holds for exchanging the body for the predicate. If only a partial permission to a predicate is held the permissions in the body will be multiplied by this value.

The explicit `fold` and `unfold` allows to specify recursive predicates. Predicates are never folded or unfolded automatically.

Functions abstract over a pure expression. They can have several parameters and have exactly one return value. As the expressions are pure they cannot change the state of the program and can therefore be used in assertions. It is necessary to specify the permissions required to evaluate the expression in the precondition of a function but as they are pure it is not possible to lose any permission. Therefore, they will always return all obtained permissions without the need to explicitly mention it in the postcondition.

Domains are a way to specify custom types and axiomatize a mathematical domain. One example is the encoding of a linked list [5]. A domain consists of several functions and axioms. The functions and axioms are global and can for example also mention or refer to functions from other domains.

Methods are the core part of a Viper program as they specify the program behavior which is verified. A method can have several parameters as well

```
field f: Int

predicate pred(x: Ref, i: Int)
{
  acc(x.f, 1/2)
}

method client(x: Ref)
  requires pred(x, 1)
  ensures pred(x, 2)
{
  var x: Int
  assert acc(pred(x, 1)) // ok
  assert acc(pred(x, 2)) // fails, pred(x, 1) != pred(x, 2)
  x := x.f               // fails, no permission to x.f
  unfold pred(x, 1)      // trade for predicate body
  x := x.f               // ok
  assert acc(pred(x, 1)) // fails, was exchanged
  fold pred(x, 2)        // trade for predicate
  x := x.f               // fails again
  assert acc(pred(x, 1)) // fails, pred(x, 1) != pred(x, 2)
  assert acc(pred(x, 2)) // ok
}
```

Listing 5: Example of predicates in Viper and how they work.

as several return values. Furthermore, pre- and postconditions can be specified (the example in Listing 5 specifies that a full permission to the predicate 'pred' is required and ensured). At the beginning of a method all preconditions are inhaled (i.e., assumed and permissions are added) and at the end all postconditions are exhaled (i.e., asserted and permissions are removed).

The body of a method is a series of statements. Most statements are well known from regular imperative languages: local variable declarations, variable assignments, conditionals, while loops and method calls. Other possible statements which are Viper specific are inhales, exhales or the previously mentioned folds and unfolds.

The `inhale` and `exhale` statements are used to manage permissions. For example, `inhale acc(x.f, 1/2)` will add half a permission to the memory location pointed to by `x.f`.

As mentioned before all methods are verified in isolation. If another method is called its body is ignored and only the contracts are considered. Upon a method call the preconditions will be exhaled and the postconditions will be inhaled. It is also possible to specify methods without a body to model calls to external code, for example a system call or a call into a library.

Expressions are similar as in many well known imperative languages. Any boolean typed expression can be used as an assertion in Viper. Additionally, assertions can also contain additional, Viper specific features such as access predicates, implication, quantifier, `perm` and `old` expressions. Implications are a boolean assertion of the form `a ==> b` where `a` is a boolean expression and `b` is an assertion. A `perm` expression allows to access the current permission amount which is held to a certain location. For example, `perm(x.f)` will return the held permission amount to the field f on object x. Old expressions allow to refer to a previous heap state: `old(expression)` will evaluate the expression in the heap state at the beginning of the method. `old[labelname](expression)` refers to the heap state at the specified label. A label is written as `label labelname` and can be inserted anywhere in the body of a method.

Two kinds of quantifiers can be used in assertions: `forall a: Type, ... :: assertion` and `exists a: Type, ... :: assertion`. The first one, a universal quantifier, evaluates to true if the assertion is true for all possible values of the specified variables and the existential quantifier will be true if at least one possible assignment of the specified variables makes the assertion true. Universal quantifiers are often used in combination with a set or sequence in the form `forall r: Ref :: r in s ==> acc(r.f)`. Note that `forall r: Ref :: acc(r.f)` is equivalent to false as `null` is included in the quantification and it is not possible to hold any permission to `null.f`.

Most statements of Viper can be represented by explicit inhales and exhales. For example, a method call is semantically equivalent to exhaling the callee's precondition and inhaling the postcondition.

Viper already contains an approach to constrain read permissions by using so-called constraining blocks. It is very easy to use them in an unsound way and therefore, they are rarely used. It is similar to the encoding we will describe in Section 3.2.1. The constraining block is used in Chalice2Viper to constrain read permissions in method calls.

2.3 Nagini

Nagini is a front-end for Viper to verify concurrent, statically-typed Python programs. The program under verification is encoded to a Viper program and then verified using one of the back-ends available for Viper. Nagini provides a library enabling a specification language similar to the one used by Viper. This specification languages lifts many concepts such as pure functions and predicates to Python.

2.4 ARP in Chalice

Chalice [11] is a verification language based on implicit dynamic frames. It uses access predicates and it already supports ARP in some cases. The encoding for ARP used by Chalice [10] reorders assertions to better constrain used read permissions. If an assertion is inhaled or exhaled, the occurring access predicates are split into three groups. The first group does not contain `rd`. The second group are access predicates where `rd` does occur but only in positive positions. The third group consists of those access predicates where `rd` occurs but in a negative position.

The access predicates are then processed in order of their group. The first group can be exhaled/inhaled as normal. For the second group `rd` will be constrained. We consider an access predicate `acc(x.f, p)` being exhaled where `p` is the permission expression containing `rd`. `p` can be rewritten as `p' + n * rd` such that `p'` does not contain any `rd` anymore. A constraint for `n * rd < perm(x.f) - p'` is then emitted. The third group where `rd` occurs in a negative position is processed last and does not emit any new constraints.

In order to still be able to verify some examples where `rd` occurs in a negative position an ad-hoc solution is used where the read permission is constrained to be smaller than a very small constant. While this does allow to verify some examples it is for example not possible to encode the first example from the 2014 FTfJP paper by Boyland et al. [4]. This example uses a lock invariant of `1 - n * rd` to model a read/write lock.

The next chapter describes our new approach which is able to constrain `rd` in negative positions and does not rely on such ad-hoc solutions.

Chapter 3

Methodology

This chapter describes our new encoding of ARP. In Section 3.1 we show new syntax which is supported by the extended Viper language. We present two different encodings to support ARP. The simple encoding is described in Section 3.2.1. This encoding is in most cases faster than the second one but it can only handle some specific cases. Section 3.2.2 describes our new log-based approach which has much less restrictions. Section 3.3 explains how quantified permissions can be constrained. To support the new approach it is necessary to desugar parts of the Viper program. Section 3.4 shows how certain Viper constructs can be desugared in order to enable our encoding.

3.1 Extended Viper Language

The extended Viper language which we implemented in this project adds support for several types of abstract read permissions. Supported are a read qualifier (written as `rd`), counting permissions (written as `rdc(n)` for a positive integer `n`), wildcard permissions (written as `rdw`) and token permissions (written as `rd_token(tk)` and `rd_token_fresh(tk)` where `tk` is the token used). The read qualifier and counting permissions have already been discussed in Chapter 1. Wildcard and token permissions are described in the next paragraphs.

A wildcard permission denotes any positive amount which is small enough such that no information needs to be havocked (e.g., in a method call it is smaller than the currently held permission by the caller). For each occurrence of a wildcard permission a new value is used and thus it can be constrained without the danger of becoming unsound (Section 3.2.3 describes why this is the case). The downside is that it is not possible to give back the same permission amount which was transferred in the precondition. If the precondition requires a wildcard permission amount to some location and the postcondition gives it back, the caller cannot assume that those two amounts are the same.

Therefore, the caller 'loses' some permission and is not able to get back the whole permission.

Viper does not provide primitives for forking and joining threads. However, forking and joining can easily be encoded. When a method is forked, its precondition is exhaled. Once the thread is joined, the postcondition of the method is inhaled. In order to be able to get back the whole permission once a method is joined it is necessary to use the same value for `rd` in the precondition and the postcondition. This poses a problem if the thread is not joined in the same method as it was forked because the methods are verified separately. In order to solve this, it is necessary to associate the used read permission with the thread and provide syntax to refer to this permission amount. This concept is called a token permission. The thread instance is represented by a token and the read permission associated with the token is represented as `rd_token(tk)` where `tk` is the token (represented by a reference). When a thread is forked `rd_token_fresh(tk)` is used to be able to constrain the value. Later only `rd_token(tk)` may be used. The reasoning behind this distinction is explained in Section 3.2.3.

`rd` used in a predicate always refers to the same value which is represented as a constant parameterless function `globalRd`. To refer to the read permission used in predicates, a function `globalRd` can be used explicitly. In predicates the normal `rd` can be used and it will refer to this function. The short form `globalRd` (without parentheses) is available as well.

For functions the already present **`wildcard`** qualifier can be used as functions will always return all received permission and therefore it is not necessary to be able to refer to the permission value in the postcondition.

3.2 Encoding

3.2.1 Simple ARP Encoding

A simple encoding of ARP is to add an assumption stating that the permission expression containing `rd` is positive and if some permission is held it is less than the currently held permission. This assumption is added directly before a method call or a while loop which uses ARP. So-called ghost parameters are used to specify the value to use for `rd` in the callee. Ghost parameters are parameters which are only used for verification but do not influence program behavior. This encoding is a simplified version of the encoding presented in the 2013 VMCAI paper by Heule et al. [10].

This works in some cases (see Listing 6 and 7). In the example the callee specifies an access permission of `acc(x.f, rd)`. The caller successfully constrains the read permission used in the call to be smaller than the currently held permission. But the encoding can be unsound in some other cases where `rd` is in

```
method callee(x: Ref)
  requires acc(x.f, rd)
  ensures acc(x.f, rd)

method client(x: Ref)
  requires acc(x.f, rd)
  ensures acc(x.f, rd)
{
  callee(x)
}
```

Listing 6: Example we use for the simple ARP encoding. `client` and `callee` both specify a read permission to `x.f`. Upon calling `callee` the calling method will not give up all permission it holds.

```
method callee(x: Ref, callee_rd: Perm)
  requires acc(x.f, callee_rd)
  ensures acc(x.f, callee_rd)

method client(x: Ref, client_rd: Perm)
  requires acc(x.f, client_rd)
  ensures acc(x.f, client_rd)
{
  var call_rd: Perm
  assume none < call_rd
  assume none < perm(x.f) ==> call_rd < perm(x.f)
  callee(x, call_rd)
}
```

Listing 7: Encoding for Listing 6. The read permission which is used for the call is constrained to be smaller than the currently held amount.

a negative position (see Listing 8 and 9). In this example the callee specifies a precondition of `acc(x.f, rd) && acc(x.f, 1/2 - rd)` which results in a total required permission of `1/2`. The caller only holds a permission of `1/4` and thus should not be able to call the method. The emitted constraints are then (slightly simplified) `call_rd < perm(x.f)` and `1/2 - call_rd < perm(x.f)`. As `perm(x.f)` is `1/4` we have `call_rd < 1/4` and `1/2 - 1/4 < call_rd` and thus `call_rd < 1/4 < call_rd` which is false.

After false is assumed everything can be proven: $a \rightarrow b \iff \neg a \vee b$ and thus false $\rightarrow$ anything $\iff$ $\neg$false $\vee$ anything $\iff$ true $\vee$ anything $\iff$ true. Hence, as soon as false is assumed the verification becomes unsound as wrong inputs can be ”proved” to be correct.

```
method callee(x: Ref)
  // should be equivalent to acc(x.f, 1/2)
  requires acc(x.f, rd) && acc(x.f, 1/2 - rd)
  ensures acc(x.f, rd) && acc(x.f, 1/2 - rd)

method client(x: Ref)
  requires acc(x.f, 1/4)
  ensures acc(x.f, 1/4)
{
  // this call should not verify as not enough permission is held
  callee(x)
}
```

Listing 8: Example which is unsound if encoded with the simple ARP encoding.

There are some properties which guarantee that a constraint system stays sound [4]: the variables in constraints have to be ordered and variables may only be constrained by a larger variable according to the partial order (see Section 3.2.3 for details). If the above encoding is used for non-trivial expressions (e.g., where rd is in a negative position) the ordering condition may be violated. The encoding we describe in Section 3.2.2 will use these properties.

If `rd` is only used positively and in isolation (i.e., not in a sum or multiplication, e.g., `acc(x.f, rd)`) this encoding is sound as the value being constrained is always newer than the value being used to constrain from above. If a variable is newer it can always be placed below all older variables in the partial order. Thus, the ordering condition can be maintained. For a more detailed explanation of the soundness of this approach we refer to the 2013 VMCAI paper by Heule et al. [10].

3.2.2 Log-based ARP Encoding

We present now an encoding which builds on the ordering constraint and allows to encode ARP in many interesting uses. The encoding was initially sketched by Vytautas Astrauskas [1]. All further explanations in this Section 3.2 refer to this log-based approach. The basic idea is to use a log to keep track of how a permission is constructed (i.e., what part are constants or some kind of ARP). If values are only constrained by larger values according to some partial order it is guaranteed that the constraints stay sound [4]. By using a log to keep track of the permission composition it can be checked that all assumptions observe the ordering constraint and are sound. How exactly the value can be constrained is explained in Section 3.2.5. Unfortunately, to be able to accurately update the log and constrain ARP, many Viper constructs have to be desugared (see Section 3.4).

```
method callee(x: Ref, callee_rd: Perm)
  requires acc(x.f, callee_rd) && acc(x.f, 1/2 - callee_rd)
  ensures acc(x.f, callee_rd) && acc(x.f, 1/2 - callee_rd)

method client(x: Ref, client_rd: Perm)
  requires acc(x.f, 1/4)
  ensures acc(x.f, 1/4)
{
  var call_rd: Perm
  assume none < call_rd
  assume none < perm(x.f) ==> call_rd < perm(x.f)
  assume none < perm(x.f) ==> 1/2 - call_rd < perm(x.f)
  // equivalent to
  // call_rd < 1/4
  // 1/2 < 1/4 + call_rd <==> 1/4 < call_rd
  // thus we have call_rd < 1/4 < call_rd <==> false
  // therefore, the call verifies despite not having enough permission
  callee(x, call_rd)
}
```

Listing 9: Encoding for Listing 8. As the read permission is in a negative position it is constrained from above and from below which results in assuming false.

```
method callee(x: Ref)
  requires acc(x.f, write - rd) && acc(x.f, rd)
  ensures acc(x.f, write - rd) && acc(x.f, rd)

method client(x: Ref)
  requires acc(x.f)
  ensures acc(x.f)
{
  callee(x)
}
```

Listing 10: In this example it is necessary to know the composition of the current permission in order not to inhale an unsound assumption.

The example in Listing 10 shows an example where the verification becomes unsound if the composition of the current permission is not taken into account. After having constrained `none < write - rd` and exhaling the first conjunct, the method `client` is left with a permission amount of `rd`. For the second conjunct a constraint of `rd < perm(x.f) == rd` is introduced. This is equivalent to `rd < rd` which in turn is equivalent to `false`. The second constraint cannot be added without being unsound. It is therefore not always possible to constrain everything and be sound at the same time.

To be able to keep track of the composition of a permission a log is kept. The log will be updated each time some permission is added or removed. In order to correctly log permissions and constrain ARP soundly it is necessary to desugar many Viper constructs into explicit inhales and exhales (see Section 3.4). The next paragraphs will explain why desugaring and log-keeping are necessary in order to soundly verify some examples.

```
1  method callee(x: Ref)
2    requires acc(x.f, 1/4 - rd) && acc(x.f, 4 * rd)
3    ensures acc(x.f, 1/4 - rd) && acc(x.f, 4 * rd)
4
5  method client(x: Ref)
6    requires acc(x.f, 1/2)
7    ensures acc(x.f, 1/2)
8  {
9    // assume none < 1/4 - rd
10   // assume 4 * rd < perm(x.f)
11   callee(x) // method call requiring desugaring
12 }
```

Listing 11: In order to verify this example it is necessary to desugar the method call into explicit inhales and exhales.

```
1  method client(x: Ref)
2    requires acc(x.f, 1/2)
3    ensures acc(x.f, 1/2)
4  {
5    // assume none < 1/4 - rd
6    exhale acc(x.f, 1/4 - rd)
7    // assume 4 * rd < perm(x.f)
8    exhale acc(x.f, 4 * rd)
9    inhale acc(x.f, 1/4 - rd)
10   inhale acc(x.f, 4 * rd)
11 }
```

Listing 12: Example from Listing 11 desugared. The method call is replaced by two exhales for the precondition and two inhales for the postcondition.

Listing 11 shows an example where it is not possible to constrain `rd` correctly without desugaring. Without desugaring all constraints have to be applied before the call in line 11. The first conjunct results in `none < 1/4 - rd` which is equivalent to `rd < 1/4`. The second conjunct gives us `4 * rd < 1/2` or `rd < 1/8`. In total a permission of `1/4 - rd + 4 * rd == 1/4 + 3 * rd` will be exhaled. The constraint of `rd < 1/8` is not enough to be able to verify the example as the permission held in the method `client` is smaller than the maximal possible permission amount being exhaled: `1/2 == 4/8 < 1/4 + 3 * 1/8 == 5/8`.

To be able to sufficiently and soundly constrain the read permission it is necessary to add constraints between exhaling the two conjuncts (see Listing 12). The first constraint remains `none < 1/4 - rd`. Then `acc(x.f, 1/4 - rd)` is exhaled and the method is left with `1/4 + rd`. For the second conjunct the constraint is now `4 * rd < 1/4 + rd` or `rd < 1/12`. It is now possible to verify the example: `1/2 == 6/12 >= 1/4 + 3 * 1/12 == 6/12`. If the `exhale` is part of a method contract it is not possible to add new constraints between the conjuncts.

For each verification scope a new permission typed local variable is created. Each usage of `rd` is replaced by this variable. As we will see it is possible to add constraints for this value each time it is used. For methods this variable is added as an additional parameter because conceptually the value is defined by the caller of the method. A new variable will be used for while loops and call sites of methods.

To encode counting permissions a constant parameterless function `epsilonRd` is used to represent the unit permission. `rdc(n)` will be translated to `n * epsilonRd()`. For each occurrence of a wildcard permission `rdw` a new variable is created and constrained. This new variable is only used once and can never be reused.

3.2.3 Constraint System

As mentioned above, if all used variables are ordered the constraint system is guaranteed to stay sound if variables are only constrained by variables which are larger according to a chosen partial order [4]. This prevents constraints which are not satisfiable like for example $a < a$ or $a < b \wedge b < c \wedge c < a$.

Hence, before a constraint `rd < perm(x.f)` can be inhaled it has to be checked that `perm(x.f)` contains at least one part which is larger than the rd which is being constrained. For this the composition of the currently held permission has to be known. A log is used which stores the composition of all permissions to all locations.

For the log we use a list of tuples `<reference: Ref, field_id: Int, level: Int, permission: Perm>`. The reference represents the object on which the field is accessed. Each field has a unique identifier which is modeled as global, unique integer constants. The level is used to ensure that the partial order is obeyed. The permission represents the amount of permission which is added/

removed to the location for the given level. If a permission consists of parts of different levels, the parts have to be logged separately on their corresponding level. Permissions will be logged as a positive amount for an `inhale` and as a negative amount in an `exhale`. Thus, to get the current permission amount for a certain location and level, all entries corresponding to this location and level can be summed up.

Before constraining a variable it has to be made sure the currently held permission contains variables which are on a higher level than the variable we try to constrain. To check this, the log can be summed up using only entries with a higher level than the level of the variable being constrained. If this is the case a constraint may be emitted.

Table 3.1 shows the levels which are used to store an added/removed permission of a certain type as well as the levels which are checked before constraining the variable [1]. The check ensures that the current permission contains a larger variable and thus, the ordering condition is preserved. There may be several variables on the same level but due to the levels which are checked the partial order is still observed.

As an example we take an access predicate `acc(x.f, rd)` from a precondition of a method call. As it is a fresh value the levels listed for the category FRESH have to be used. The check to constrain the read permission would look similar to `ARPLog_sum_gt(x, field_f(), 1, log) > none ==> call_rd < perm(x.f)` where `field_f()` is a unique identifier for the field, `log` is the log of the current scope and `ARPLog_sum_gt` is a function summing up all values in the log for the given reference and field with a level strictly greater than the specified value, which in this case is level 1. The permission change would be logged as `log := ARPLog_Cons(x, field_f(), -call_rd, 1, log)`. Because the read permission is exhaled at the call site it is subtracted and thus logged as a negative amount.

An occurrence of `rd_token_fresh(tk)` will be translated to `rd_token(tk)` but it is constrained based on level FRESH instead of level token. This can be done because it is a new value which cannot already be constrained by other values. Later the token permission might be older than the context's read permission and it therefore has to be constrained according to a higher level than CONTEXT. `rd_token_fresh(tk)` has to be logged on level TOKEN as all further usages of the value will be logged on level TOKEN as well. Not logging them on the same level might result in unsoundness (note that `rd_token_fresh(tk)` and `rd_token(tk)` refer to the same permission amount): after (`write`, CONST), (`-rd_token_fresh(tk)`, FRESH), (`rd_token(tk)`, TOKEN), (`-write`, CONST) has been logged for some location it is possible to add a constraint `rd < none` for a fresh `rd` because the log summed up above level FRESH will return `rd_token_fresh(tk)`. Logging `rd_token_fresh(tk)` on level TOKEN as well prevents this problem.

Category	**Store**	**Check**	**Description**
CONST	6	>5	Constant permission amount
GLOBAL	5	>5	Read permission used in predicates
CONTEXT	4	>4	Read permission used in the surrounding scope (e.g., rd used in preconditions come from the calling scope)
TOKEN	3	>4	Token permissions. As they can be older than permissions from the surrounding scope they cannot be constrained by CONTEXT level variables. But as they might be newer CONTEXT cannot be constrained with a token permission
WILDCARD	2	>0	Wildcard permissions are only used once. Therefore they cannot be constrained further. Thus they can also be constrained by other wildcard permissions as they are all older and therefore larger in the partial order
FRESH	1	>1	A new rd permission (e.g., for a method call or a loop). As shown in Listing 10 a fresh variable may not be constrained with itself
EPSILON	0	>0	Counting permission

Table 3.1: Log levels. Store represents the level on which a value of the given category has to be stored in the log. Check specifies which levels are checked before a constraint may be emitted.

For each verification scope a new log has to be started. In Viper a new log is needed for each method and in each while loop. If the while loop uses an ARP in its invariant it is logged on the level FRESH in the surrounding scope's log but on the level CONTEXT within the while loop.

3.2.4 Log Update

A log update has to be done for all statements which change the permission amount of a location. In Viper those are inhales, exhales, folds, unfolds, method calls, loops and method contracts.

In order to log the permissions on the correct levels, the permission expression first has to be brought into a normalized form $q + n_g * \texttt{globalRd}() + n_{\texttt{tk1}} * \texttt{rd\_token}(tk1) + n_{\texttt{tk2}} * \texttt{rd\_token}(tk2) + \cdots + n_{\texttt{rd}} * \texttt{rd} + n_{\texttt{cnt}} * \texttt{epsilonRd()}$ where q is a constant permission, `globalRd()` represents a read permission in a predicate, `rd_token(tk1)` is the read permission which was passed to the forked method being represented by token `tk1`, `rd` is the read permission and `epsilonRd()` is the unit used in counting permissions. For now we assume such a normalized form.

For each level of the normalized expression a separate log update is performed. We assume for the moment that access predicates are not in a quantified expression and do not contain any **perm** expressions. We will revisit the quantified case in Section 3.3. In this simple case the update can just be added to the front of the list. If a permission is added it is logged positively, if the permission is removed it is logged negatively. Listing 13 shows the log updates for a method call which specifies `acc(x.f, write - rd)` for pre- and postcondition. In the example the call has been replaced by explicit inhales and exhales of the method contracts. `rd` in the contracts is replaced by the actual value `call_rd` which would be constrained before the shown part.

```
log := ARPLog_Cons(x, field_f(), -write, 6, log)
log := ARPLog_Cons(x, field_f(), call_rd, 1, log)
exhale acc(x.f, write - call_rd)
inhale acc(x.f, write - call_rd)
log := ARPLog_Cons(x, field_f(), write, 6, log)
log := ARPLog_Cons(x, field_f(), -call_rd, 1, log)
```

Listing 13: Log update for an **inhale** and **exhale** of `acc(x.f, write - rd)`.

In order to also log permissions to predicates each distinct predicate also needs a unique identifier. This is implemented using functions for each predicate which take the same parameters as the predicate. Using axioms it is ensured that the same predicate with different parameters or a different predicate have different identifiers. As predicates do not have a receiver, `null` can be used for the reference part in the log.

3.2.5 Constraining

Before an ARP is used in an access predicate – this might be in an `inhale`, `exhale` or an `assert` – the values of the used ARP constructs have to be constrained. As we saw above the constraints stay sound by only constraining a variable by a larger variable according to the partial order. The partial order used is based on the check value of Table 3.1. `none` is the lowest value in the order and all other constant values are larger than any variable.

We assume a normalized form of the permission as described in Section 3.2.4. It holds that $\forall q, n_g, n_{\texttt{tk}}, tk, n_{\texttt{rd}}, n_{\texttt{cnt}}.\, \texttt{none} < n_{\texttt{cnt}} * \texttt{epsilonRd()} < n_{\texttt{rd}} * \texttt{rd} < n_{\texttt{tk}} * \texttt{rd\_token}(\text{tk}) < n_g * \texttt{globalRd()} < q \leq \texttt{write}$. For all occurring parts a constraint is added which ensures this order. For example, for a permission expression of `1/2 + 4 * rd + rdc(3)` the constraint `3 * epsilonRd() < 4 * rd && 4 * rd < 1/2` is emitted. The partial order is trivially followed and thus, this constraints can be added without any checks. This already ensures simple properties like for example $\texttt{none} < \texttt{rd} - \texttt{rdc(1)} < \texttt{write}$.

The goal of the added constraints is to make sure the whole expression is less than the currently held permission amount and larger than none. To be as complete as possible while still being sound, as much of the expression as possible should be part of the constraint. For example, if `rd + rdc(1)` is constrained, the constraint `rd + rdc(1) < perm(...)` is more powerful than the two separate constraints `rd < perm(...)` and `rdc(1) < perm(...)`. The following encoding ensures that as many parts as possible are part of the emitted constraint.

The resulting assertion will be of the form `C ==> A < perm(x.f)` where `C` is a check on the permission composition and `A` is a sum of ARP parts. Starting from the smallest occurring ARP part it is checked whether the part is in a positive position and the current permission contains a larger variable. If this is the case the part is added to `A`. Otherwise it is omitted and the process is continued with the next larger part. As soon as the current permission does not contain a larger part any more but it is in a positive position the collected assumption can be emitted. Constants are a special case as it has to be checked that the constant part of the currently held permission is larger than the constant occurring in the expression being processed.

If some part is in a negative position the largest non-zero part has to be positive. Because all remaining parts are smaller than the found positive part we assume that the whole expression is larger than zero. If the largest non-zero part is negative it is not possible to soundly constrain the values.

Listing 14 shows an example of the constraints which are added for an **exhale** of **acc**(x.f, 1/2 + 4 * rd + rdc(7). The function `ARPLog_sum_gt` sums up the values of `ARPLog_sum` for all levels which are strictly greater than the specified level. For this example it is assumed that the **exhale** is from a desugared method call and thus `rd` will be on level FRESH. `rdc(7)` has been translated to `7 * epsilonRd()` and `field_f()` is a function which returns a unique identifier for the field `f`. For clarity the assumption has been split into several lines. The first two **assume** on lines 1 and 2 ensure the order of the different levels. The third **assume** starting from line 3 checks the log and assumes suitable assumptions.

```
1   assume 7 * epsilonRd() < 4 * rd // level order
2   assume 4 * rd < 1 / 2 // level order
3   assume ( // check log and assume
4     none < ARPLog_sum_gt(x, field_f(), 0, log) ? // level EPSILON
5       (
6         none < ARPLog_sum_gt(x, field_f(), 1, log) ? // level FRESH
7           (
8             none <= 1 / 2 ==>
9               (
10                1 / 2 < ARPLog_sum_gt(x, field_f(), 5, log) ? // level CONST
11                  (
12                    1 / 2 + (4 * rdr + 7 * epsilonRd()) < perm(x.f)
13                  ) : (
14                    4 * rd + 7 * epsilonRd() < perm(x.f)
15                  )
16              )
17          ) : (
18            none <= 1 / 2 ==>
19              (
20                1 / 2 < ARPLog_sum_gt(x, field_f(), 5, log) ? // level CONST
21                  (
22                    1 / 2 + 7 * epsilonRd() < perm(x.f)
23                  ) : (
24                    7 * epsilonRd() < perm(x.f)
25                  )
26              )
27          )
28      ) : (
29        none < ARPLog_sum_gt(x, field_f(), 1, log) ==> // level FRESH
30          (
31            none <= 1 / 2 ==>
32              (
33                1 / 2 < ARPLog_sum_gt(x, field_f(), 5, log) ? // level CONST
```

```
                    (
                      1 / 2 + 4 * rd < perm(x.f)
                    ) : (
                      4 * rd < perm(x.f)
                    )
               )
          )
     )
)
exhale acc(arg_x.f, 1 / 2 + 4 * rd + 7 * epsilonRd())
```

Listing 14: Constraining ARP values for an **exhale** of **acc**(arg_x.f, 1 / 2 + 4 * rd + 7 * epsilonRd()). The log is checked to contain a larger value before a constraint is applied.

An example where some parts might be in a negative position is shown in Listing 15. As part of a desugared method call the expression **acc**(x.f, 1/2 + n * rd + rdc(nc)) is exhaled for some integer parameters n and nc. The first two **assume** are again a constraint for the order of the levels. Due to the fact that n might be negative the absolute value is considered.

```
assume (nc < 0 ? -nc : nc) * epsilonRd() < (n < 0 ? -n : n) * a_rd
assume (n < 0 ? -n : n) * a_rd < 1 / 2

assume (
  0 <= nc ?
    (
      none < (ARPLog_sum_gt(x, field_f(), 0, log)) ?
        (
          0 <= n ?
            (
              none < (ARPLog_sum_gt(x, field_f(), 4, log)) ?
                (
                  1 / 2 < (ARPLog_sum_gt(x, field_f(), 5, log)) ?
                    (
                      1 / 2 + (n * a_rd + nc * epsilonRd()) < perm(x.f)
                    ) : (
                      n * a_rd + nc * epsilonRd() < perm(x.f)
                    )
                ) : (
                  1 / 2 < (ARPLog_sum_gt(x, field_f(), 5, log)) ?
                    (
                      1 / 2 + nc * epsilonRd() < perm(x.f)
                    ) : (
            nc * epsilonRd() < perm(x.f)
           )
         )
       ) : (
        (
         none < 1 / 2 + (n * a_rd + nc * epsilonRd())
        ) && (
         1 / 2 < (ARPLog_sum_gt(x, field_f(), 5, log)) ?
          (
           1 / 2 + nc * epsilonRd() < perm(x.f)
          ) : (
           nc * epsilonRd() < perm(x.f)
          )
        )
       )
     ) : (
      0 <= n ?
       (
        none < (ARPLog_sum_gt(x, field_f(), 4, log)) ==>
         (
          1 / 2 < (ARPLog_sum_gt(x, field_f(), 5, log)) ?
           (
            1 / 2 + n * a_rd < perm(x.f)
           ) : (
            n * a_rd < perm(x.f)
           )
         )
       ) : (
        none < 1 / 2 + n * a_rd
       )
     )
   ) : (
    (
     0 < n ?
      (
       none < 1 / 2 + nc * epsilonRd()
      ) : (
       0 == n ?
        (
         none < 1 / 2 + nc * epsilonRd()
        ) : (
         none < n * a_rd + (1 / 2 + nc * epsilonRd())
        )
      )
      ) && (
        0 <= n ==>
          (
            none < (ARPLog_sum_gt(x, field_f(), 4, log)) ==>
              (
                1 / 2 < (ARPLog_sum_gt(x, field_f(), 5, log)) ?
                  (
                    1 / 2 + n * a_rd < perm(x.f)
                  ) : (
                    n * a_rd < perm(x.f)
                  )
              )
          )
      )
    )
)

exhale acc(x.f, 1 / 2 + n * a_rd + nc * epsilonRd())
```

Listing 15: Constraining ARP values in negative positions for an `exhale` of `acc(x.f, 1 / 2 + n * a_rd + nc * epsilonRd())`. To constrain a variable in a negative position it is not necessary to check the log as long as there is a larger variable in a positive position.

3.3 Quantified Permissions

If access predicates are used in a universal quantifier, updating the log and constraining read permissions is more difficult than in the non-quantified variant. As not all affected locations are explicitly enumerated, updating and constraining has to be done in a quantifier as well.

All quantifiers which contain access predicates can be rewritten in the form `forall r: Ref, ... :: A ==> acc(B, C)` for some quantified variables, a condition `A`, a location `B` and a permission amount of `C`. This transformation is applied by Viper upon parsing the input.

In the simple encoding handling ARP in quantified expressions is relatively easy. The quantifier can just be used with the exact same quantified variables and the access predicate is replaced by the constraint. In Listing 16, an example of the encoding is shown. There are no limitations imposed by the encoding on any part of the quantifier.

In the log-based encoding quantified permissions pose a much bigger challenge. To log quantified expressions it is not possible to enumerate all occurring permission updates and add an entry for each of them – there might

```
method callee(s: Seq[Ref], callee_rd: Perm)
  requires forall r: Ref :: r in s ==> acc(r.f, callee_rd) && r.f > 0
  ensures forall r: Ref :: r in s ==> acc(r.f, callee_rd) && r.f > 0

method client(s: Seq[Ref, client_rd: Perm)
  requires forall r: Ref :: r in s ==> acc(r.f, client_rd) && r.f > 0
  ensures forall r: Ref :: r in s ==> acc(r.f, client_rd) && r.f > 0
{
  var call_rd: Perm
  assume none < call_rd
  assume forall r: Ref :: r in s ==>
    none < perm(r.f) ==> call_rd < perm(r.f)
  callee(s)
}
```

Listing 16: Quantified ARP in the simple encoding. The access predicate in the quantifier is replaced by the constraint for the read permission.

be an infinite number of updates necessary. For summing up the permissions in the log a function `function ARPLog_sum(ref: Ref, fieldId: Int, level: Int, log: ARPLog): Perm` is used, which sums all entries with the corresponding reference, field and level. To check permissions before constraining the function `ARPLog_sum_gt` sums up `ARPLog_sum` for all needed levels.

The encoding described below can update the log only if the only quantified variable is a reference and `B` in the mentioned normalized quantifier is of the form `r.f` for the quantified reference `r` and some field `f`. Access to predicates and more complex location expressions is therefore not supported in a quantified assertion. There are no limitations for `A` or `C`.

To update the log a new log variable is created for which no information is present. Using a `forall` expression, information from the old log is transferred to the new log and updated where needed. For locations affected by the quantifier, the new log is changed accordingly, and for all others it stays unchanged. To transfer the information we quantify over all parameters of the `ARPLog_sum` function except the log. If condition `B` holds for the quantified reference, the permission amount corresponding to the quantified level is added or subtracted. Otherwise, the old value is kept. Listing 17 shows an example of a quantified log update. For clarity the `assume` has been split into multiple lines.

If the location in the access predicate does not conform to the described form the update of the log is much harder. To understand the difficulty we will consider Listing 18 from the Viper test suite. `IArray` is a custom domain modeling an array and `loc(a, i).val` represents the element at position `i` in

```
method client(s: Seq[Ref])
{
  var log: ARPLog := ARPLog_Nil()
  ...
  inhale forall r: Ref :: r in s ==> acc(r.f, 1/2 + rdc(1))
  var new_log: ARPLog
  assume forall ref: Ref, fld: Int, level: Int ::
      ARPLog_sum(ref, fld, level, new_log) ==
      ARPLog_sum(ref, fld, level, log) +
      (
        (fld == field_f() && ref in s) ?
        (
          level == CONST ?
            1/2
          : (
            level == EPSILON ?
              rdc(1)
            :
              none
          )
        ) :
          none
      )
  log := new_log
  ...
}
```

Listing 17: Quantified log update for a quantified access predicate. The log is only updated for references where the condition holds and all other entries stay the same. Triggers for the quantifiers were removed for simplicity.

array `a`. To log the permissions it is not only necessary to know which values changed but also which did not change. To specify the changed values the access predicate in the quantifier can be replaced by the log update which was used above. The difficulty is to specify that all permissions to other locations which are not part of the quantifier did not change.

One approach we tried is to use an existential quantifier in order to figure out which values changed. An encoding for the first quantifier in the example would look like Listing 19. Existential quantifiers are difficult for the underlying SMT solver to prove and even a simple looking expression like `forall i: Int :: exists j: Int :: i == j` might not be able to be verified. Due to this the approach does not work in practice. We were not able to verify any example using this approach.

```
method test01(a: IArray, n: Int)
  requires n > 5
  requires forall i: Int :: i in [0..n) ==> acc(loc(a, i).val)
  ensures  forall i: Int :: i in [0..n) ==> acc(loc(a, i).val)
  ensures  loc(a, 1).val == loc(a, 0).val + old(loc(a, 1).val)
{
  loc(a, 1).val := loc(a, 0).val + loc(a, 1).val
}
```

Listing 18: Array access in a quantifier. It is difficult to log such an assertion.

```
inhale forall i: Int :: i in [0..n) ==> acc(loc(a, i).val)
var new_log: ARPLog
assume forall ref: Ref, fld: Int, level: Int ::
    ARPLog_sum(ref, fld, level, new_log) ==
    ARPLog_sum(ref, fld, level, log) +
    (fld == field_val() && (exists i: Int :: i in [0..n) && loc(a, i) == ref) ?
      (level == CONST ? write : none) : none)
```

Listing 19: Log update approach for quantified array access seen in Listing 18. Triggers for the quantifiers were removed for simplicity. Due to the use of the existential quantifier it is not well suited for the underlying SMT solver.

Perm expressions can be used to refer to the currently held permission amount. If the permission to one location is added to another location (e.g., `acc(x.f, perm(x.g))`), we have to make sure we can keep the information about the composition of the permission. As the added permission can be composed of different ARP types and constants, it is necessary to transfer the logged information to the location for which the permission is changed. A similar approach as for quantified expressions can be used. Listing 20 shows an example for the encoding. Instead of the new permissions the current permission of the other location is used and the condition is that the quantified reference and quantified field are the receiver of the permission. A quantifier is needed because the number of levels might in general not be bounded (in our case they are but in a more general case they might not be).

If ARP are used in a quantified assertion it is necessary to constrain the read permissions. The approach to constraining described in Section 3.2.5 can be used to constrain values even if they occur in a quantified assertion. Contrary to the log update it is only necessary to make assumptions about the references occurring in the assertion. No assumption about references not part of the assertion are necessary. Thus, it is possible to adopt the original quantifier and replace the access predicate by the same assumption as for the non-quantified case. Because the assumption is a single assertion without the need for multiple statements this is possible to do. It would be

```
method client(a: Ref, b: Ref)
{
  var log: ARPLog := ARPLog_Nil()
  ...
  inhale acc(a.f, perm(b.f))
  var new_log: ARPLog
  assume forall ref: Ref, fld: Int, level: Int ::
      ARPLog_sum(ref, fld, level, new_log) ==
      ARPLog_sum(ref, fld, level, log) +
      (
        (ref == a && fld == field_f()) ?
          ARPLog_sum(b, fld, level, log)
        :
          none
      )
  log := new_log
  ...
}
```

Listing 20: Log update if **perm** is used in a permission. As the levels might not be bounded, a quantified log update is used.

possible to constrain ARP for arbitrary locations like for example the one shown in Listing 18. In practice this cannot be done because it is not possible to update the log and thus we risk to become unsound. If the log updating can be improved, constraining ARP in quantified assertions comes 'for free'.

To summarize, ARP are currently supported in quantified assertions only if the simple encoding is used or if the location is of the form `x.f` for a bound variable `x` and a field `f`. In this case, the permissions can be correctly logged and the ARP values are suitably constrained.

3.4 Desugaring

As described in Section 3.2.2 it is necessary to desugar Viper constructs which alter permission information including method calls and while loops into explicit inhales and exhales. This allows us to accurately update the log and emit constraints at the correct position. It is important to reproduce the same behavior in the desugared form as in its original form. This proved to be difficult and was not always possible. This section describes how the different Viper constructs can be desugared in order to be able to keep an accurate log and suitably constrain ARP. Some statements can not directly be desugared and have to be handled separately.

3.4.1 Methods

Methods must not have any contracts in order for the encoding to be able to correctly constrain read permissions as explained in Section 3.2.2. This allows us to add statements before and between contracts. Therefore, all contracts are replaced by explicit inhales for preconditions or exhales for postconditions respectively.

In order to be able to support `old` expressions, a label is inserted right after the preconditions have been inhaled. As `old` considers the permission state at the referred label position, the label is placed after inhaling the preconditions, otherwise not enough permission would be present. All `old` expressions which do not already refer to a label are rewritten to point to the newly added label.

The postconditions as well as other contracts use the permission state at the end of the method body to check location accesses. For single exhales the state just before the `exhale` is relevant. Listing 21 shows how exhaling contracts differ from exhaling normal assertions. Both times a `write` permission is inhaled and then exhaled and a statement about the value is asserted. The check whether enough permission for `a.f` is present is done for line 4 in the state at the end of the method body in line 7. Therefore, the assertion is successful. For line 15 the state just before the statement is relevant. As in line 14 the permission was already exhaled, not enough permission is left and the assertion fails due to missing permission.

Therefore, if a method has more than one postcondition it would change the semantics of the program to just add an `exhale` for each postcondition. It is therefore necessary to add a label just before exhaling the postconditions. All heap-dependent expressions in the postconditions which are not yet in an old expression are then wrapped in an old expression referring to this label.

Viper checks contracts for well-formedness. Among other things this means that the contracts are self-framing: if a location is accessed, the needed permission needs to be mentioned in the contract. If contracts are desugared into inhales/exhales this check will no longer be performed. Therefore, an additional method is added which just has the contracts but no body. For this new method well-formedness checks are performed. Reported errors will be mapped back to the original method. The method does not have a body and therefore, it is not checked whether the postcondition holds. Viper performs only well-formedness checks for these methods.

To be able to refer to the read permission which is used in the method, an additional parameter of type `Perm` is added. The new parameter is constrained to be strictly between `none` and `write`. All occurrences of `rd` in the method body are rewritten to this added parameter.

```
method contracts(a: Ref)
  require acc(a.f) && a.f == 0
  ensures acc(a.f) // remove permission
  ensures a.f == 1 // successful
{
  a.f := 1
  // end of method body
}

method explicit(a: Ref){
{
  inhale acc(a.f) && a.f == 0
  a.f := 1
  exhale acc(a.f) // remove permission
  exhale a.f == 0 // fails
}
```

Listing 21: Difference between handling of method postconditions and exhales. If the exhales are written exactly like the postconditions, the second `exhale` fails.

3.4.2 Method Calls

Method calls can be replaced by first exhaling all preconditions and then inhaling all postconditions. Old expressions in the postconditions as well as all location accesses need to be rewritten to an old expression referring to a label added just before the call. Location accesses need be rewritten using old expressions due to the same reason as mentioned in Section 3.4.1 – the permissions available in a single exhale might not be the same as when the exhale is part of a postcondition and thus processed at once.

For the method call a fresh variable for the read permission in the called method is created. Occurrences of `rd` in the method contract of the called method are replaced by this new variable.

The pre- and postconditions can refer to method parameters and return values. These have therefore to be exchanged for the actual arguments and targets which are used in the call. If an expression is used as an argument it would be unsound to just replace all occurrences of the argument in the contract with the expression used in the call (see Listing 22 and 23). Therefore, for each argument a new local variable is created and the corresponding expression is assigned. Those new variables are then used to replace the arguments. The only information which is available about the return values comes from the postconditions. Therefore, variables which are assigned to are havocked before the postconditions are inhaled.

```
method callee(a: Int) returns (b: Int)
  ensures b > a

method client()
{
  var x: Int
  x := callee(x)
}
```

Listing 22: Just replacing arguments in a method call might be unsound if a naive encoding is used.

```
method callee(a: Int) returns (b: Int)
  ensures b > a

method client()
{
  var x: Int
  // x := callee(x)
  // ensures b > a
  // a -> x, b -> x
  assume x > x // equivalent to false
}
```

Listing 23: Naive encoding of Listing 22. Because the target of the call is used as an argument it is constrained against itself.

3.4.3 While Loops

As for methods it is also necessary to remove contracts in while loops.

During verification the loop invariant and the loop condition is inhaled for the loop body. At the end of the body the invariant is exhaled. Outside of the loop in the method the invariant is exhaled before the loop and together with the negation of the loop condition inhaled after the loop.

To desugar this behavior we exhale the loop invariant before the loop. In the loop body we inhale the loop invariant and the loop condition. At the end of the loop body we exhale the loop invariant. Finally, we inhale the invariant and the negation of the loop condition after the loop body. As the loop invariant may contain more than one assertion, we wrap heap-dependent assertions in an old expression with the label being just before the loop invariants are exhaled. For the loop we create and constrain a fresh variable for read permissions used in the invariant, just as for a method call. Within the loop body we create a new log because the loop body is a new verification scope.

Due to the missing loop invariant no permissions are available for the loop condition. If the condition is heap-dependent this poses a problem. Therefore a new local boolean variable is used as a replacement. Before exhaling the invariant outside the loop as well as inside the loop the condition is assigned to the local variable. Further, at the start of the loop body the loop condition is assumed and after the loop the negation of the loop condition is assumed. Listing 24 and 25 show an example of a desugared while loop.

```
method client(x: Ref)
  requires acc(x.f) && x.f <= 5
  ensures acc(x.f) && x.f == 5
{
  while(x.f != 5)
    invariant acc(x.f) && x.f <= 5
  {
    x.f := x.f + 1
  }
}
```

Listing 24: While loop which is desugared in Listing 25.

```
method client(x: Ref)
{
  inhale acc(x.f) && x.f <= 5   // method precondition
  var while_cond: Bool
  while_cond := x.f != 5        // assign loop condition
  exhale acc(x.f) && x.f <= 5   // invariant
  while(while_cond)             // replaced loop condition
  {
    inhale acc(x.f) && x.f <= 5 // invariant
    assume x.f != 5             // loop condition
    x.f := x.f + 1              // actual loop body
    while_cond := x.f != 5      // assign loop condition
    exhale acc(x.f) && x.f <= 5 // invariant
  }
  inhale acc(x.f) && x.f <= 5   // invariant
  assume !(x.f != 5)            // loop condition
  exhale acc(x.f) && x.f == 5   // method postcondition
}
```

Listing 25: While loop from Listing 24 desugared. The loop invariants are removed and replaced by explicit inhales and exhales. Furthermore, the loop condition is replaced due to missing permissions if it is left in the **while**.

Like pre- and postconditions, Viper checks invariants for well-formedness. For invariants the contract is not considered in isolation but information available in the method at this point is taken into account. If, for example, the divisor of a division is a local variable, it will be checked that this variable is non-zero. In the method there might be information that implies that the variable cannot be zero and the verifier will not complain. When the loop is desugared, this check is not performed any more.

One approach to still perform a well-formedness check is to create an additional method with the invariant as a postcondition (as the invariant might contain old expressions it cannot be put as a precondition). This proved not to be a viable approach because it fails to verify examples which were possible before. Since less information is available, as explained above, the contract can fail the check even though in the original program there is no problem.

We decided to give up well-formedness checks for loop invariants. As everything which would lead to a not well-formed invariant also leads to another error (e.g., if the contract is not self-framing, it will result in an insufficient permission error when inhaling the loop invariant in the loop body) this change does not lead to wrong programs being verified successfully, but the type of the error will change.

3.4.4 Fold / Unfold

A fold is semantically equivalent to exhaling the predicate body and inhaling access to the predicate except that knowledge about memory locations is not havocked. An unfold corresponds to exhaling access to the predicate and inhaling its body. Due to the difference in havocking behavior the statements cannot simply be replaced by inhales and exhales. Instead the changes in permissions are directly added to the log. The log update is equivalent to the case where the folds/unfolds were replaced by the corresponding inhales/ exhales. To make sure enough permission is present to log all changes, the update will take place before a `fold` but after an `unfold`.

3.4.5 Inhale / Exhale

Inhales and exhales need to be split into multiple statements if they contain a `perm` expression or use ARP. Each conjunct of the assertion is treated separately. Assertions containing implications or ternary expressions are wrapped together with the corresponding log update in if statements. This is necessary for log-keeping as the implied expression might change the value of the expression which is being evaluated for the implication/ternary expression. E.g., after inhaling `perm(x.f) < 1/2 ==> acc(x.f, 1/2)` the condition will be false. If the implication is first inhaled and then the log update is done separately the log will not contain the added `1/2`.

If an **inhale** or **exhale** is split into several statements it is necessary to wrap all heap-dependent expressions into an old expression referring to a label just before the first of the split statements. This is similar to the wrapping needed for method contracts seen in Section 3.4.1 and Section 3.4.2.

Chapter 4

Implementation

We implemented the encoding described in Chapter 3 in Viper and made ARP available in Nagini. This chapter describes different aspects and challenges of the implementation.

4.1 Viper to Viper Translation

We have decided to not implement ARP directly in Viper but as an extended Viper language which will be encoded to normal Viper. The Viper language already supports a well-rounded set of features and constantly adding new things would bloat the core language. As there are some front-ends building on top of Viper (e.g., Nagini [7] for Python, VerCors [2] for Java) it is not desirable to keep changing it. As we have demonstrated in the previous chapter, it is possible to encode ARP in the existing Viper language and thus it was not necessary to implement them directly in Viper.

We implemented an extension to Viper which encodes the extended Viper language to the normal Viper language. The extension receives the input, a program written in an extended form of the Viper language, and produces an equivalent Viper program. This program is then verified and the resulting errors are translated back to the original program. To be able to implement such Viper-to-Viper translations in an easy and clean fashion we developed a plugin system which allows to add hooks at different stages of the parsing and translation stage. Section 4.5 introduces the main features of this system.

For the translation we used an AST [1] rewriting framework for Viper, which was developed in a Master's Thesis [9] by Simon Fritsche. This framework allows to specify a partial function mapping AST nodes to new AST nodes and it allows to keep custom context information.

[1]Abstract Syntax Tree

As described in Section 3.2, the extension adds syntax to use abstract read permissions (`rd`), counting permissions (`rdc(n)` for positive integer n), wildcard permissions (`rdw`) and token permissions (`rd_token(tk)` and `rd_token_fresh(tk)` for some reference typed token `tk`) in access predicates (e.g., `requires acc(x.f, rd)`).

To be able to parse the extended Viper language without changing the existing parser, the rewriting takes place in two phases. The first phase takes place after the parsing but just before the identifiers are resolved. In this phase functions for `rd`, `rdc`, `rdw`, `globalRd`, `rd_token` and `rd_token_fresh` are added such that they can be resolved. As it is nicer to write `acc(x.f, rd)` instead of `acc(x.f, rd())` all uses of `rd` which are not yet a function call are transformed into a call.

The second phase takes place just before the verification. The parse AST which was manipulated in the first phase has been translated into a Viper AST by Viper. This AST contains much more information than the parse AST because all variables and calls have been resolved. For example, types of variables are known and calls to methods can be distinguished from function calls. During this second phase the desugaring (see Section 3.4) and the encoding explained in Section 3.2 is performed. During this phase information used to translate errors back (see Section 4.4) is added to the AST.

For the encoding two domains are needed. The first one is the domain describing the log used to store the permissions. This domain does not depend on the program which is verified and can be loaded from file. The second domain provides unique integer identifiers for fields and predicates to be used when updating and querying the log. Each field and parameterless predicate gets a parameterless function marked as unique. Such a function is guaranteed to return a unique value which is not returned by any other function marked as unique. For each predicate with parameters, a function with the same parameters is generated. As the identifier has to be unique, some axioms have to be added. One axiom states that the function does not return the same value for two different choices of parameters. This ensures that the same predicate with different parameters can be differentiated. Additionally, for each other field and predicate an axiom is added to state that whatever the parameters, the value of the two functions will be different. Listing 26 shows an example for the generated axioms.

To be able to generate unique names for additional local variables and method arguments, all used identifiers are collected at the start of the translation. Once a new name is needed it is checked whether the desired name is already used. If it is, a number in the name will be increased until an unused name is found. This new name is then added to the list of used names. This prevents any name collisions between newly added identifiers and existing ones.

```
domain ARP_field_functions {
  unique function field_f(): Int
  function predicate_p2(x: Ref): Int
  function predicate_p3(x: Ref, y: Ref): Int

  axiom ARP_p2_f {
    (forall x: Ref :: predicate_p2(x) != field_f())
  }

  axiom ARP_p2_p3 {
    (forall x: Ref, x_1: Ref, y_0: Ref ::
      predicate_p2(x) != predicate_p3(x_1, y_0))
  }

  axiom ARP_p2 {
    (forall x: Ref, x_0: Ref ::
      predicate_p2(x) == predicate_p2(x_0) ==> x == x_0)
  }

  // p3 axioms omitted
}
```

Listing 26: Example of generated axioms to guarantee unique identifiers for fields and parameterized predicates. The example adds unique values for a field `f` and two predicates `p2(x: Ref)` and `p3(x: Ref, y: Ref)`. For brevity axioms for `p3` and triggers in quantifiers were omitted.

4.2 Normalization

The explained encoding assumes a normalized form for the transferred permissions. Viper does allow arbitrarily complex permission expressions in access predicates. Obtaining the normalized form is not always easy to do. Our implementation is able to normalize an expression if it is linear (e.g., no $rd * rd$) and ternary expressions are only used at the top level (e.g., `acc(x.f, b ? 1/2 : 1/4)` is possible but `acc(x.f, (a ? 1/4 : 1/8) + (b ? 1/2 : 1/4))` is not). If a ternary expression in the permission expression of an access predicate is encountered it is split into two separate access predicates in the two branches of an if statement.

If an expression contains a permission typed variable it is not known how the variable is composed and it might contain parts of a non-CONST level. To be able to still log them correctly it is enforced that no ARP can ever be assigned to a variable. While directly assigning an ARP to a variable (e.g., `a := rd`) is prevented during translation it could be done by using `perm` on a field to which an `rd` permission is held. If this pattern of assigning `perm` to a variable is found an assertion is added which checks the log to make sure the assigned permission is only on the CONST level.

4.3 Optimizations

The presented log-based encoding introduces some overhead for example due to the log updates. It is possible to optimize different parts of the encoding during translation in order to minimize the introduced overhead.

To improve the performance of verification the translation can already perform some optimizations. A simple optimization is to simplify expressions in emitted statements when possible. For example, `1 * n` can be replaced by `n` and `-(-n)` is the same as `n`. While this might not make a difference as the back-end can do such optimizations as well it simplifies other optimizations described below.

One large optimization is to remove branches if they cannot be reached. One example is the check of `0 < 1` which is often generated to check whether in an expression like `1/2 + rd` the `rd` is in a positive position. While the back-end will figure out the false branch is not reachable, it is faster to already drop the branch when generating the translation. This is because the back-end will perform SMT queries to check for satisfiability but during the generation of the branch some simple cases can already be checked – like for example if both values are integer literals the branch to take can already be decided and only this branch will be emitted.

Always splitting all inhale/exhale statements turned out to be a performance problem in the VCG back-end as each performed inhale/exhale introduces some performance overhead. Therefore, those statements are only split if it is absolutely necessary.

Another optimization is to mix the encoding based on logs described in Section 3.2.2 and the simple encoding described in Section 3.2.1. In a first step it is determined which methods do not use complicated ARP expressions (that is, expressions where no ARP is used or only in the form `acc(loc, rd)` for some location `loc`) in the method contract. Each method which either uses complicated ARP expressions in the method contract or body or calls a method which uses them in the contract has to be encoded using the log-based approach. All other methods can use the simple encoding which does not require to desugar any statements.

As updating the log for each field comes with a performance penalty the user can exclude some fields if it is known that `rd` is never used with them. Listing 27 shows an example how this can be used. This syntax was chosen to be able to parse it without changing the parser.

```
field f: Int
field g: Int
field h: Int

method ARP_IGNORE(){
  g()
  h()
}
```

Listing 27: The dummy method `ARP_IGNORE` allows to specify some fields which will be ignored for the encoding. In this example fields g and h are ignored.

4.4 Error Back Translation

Because the program is heavily modified before it is verified the found errors will be at different positions in the AST, for different nodes or even of a different type. For example, the verifier may return an exhale failed error when it would have returned a precondition violated error for the original program. The Viper AST already allows to add error transformation information directly to nodes. After the verification step the back-translation can be triggered by calling a single method on each of the reported errors. This method will transform the node and change the type of the error according to the attached transformation information.

During translation of the extended Viper program to normal Viper it has to be ensured that all nodes have the correct error transformation information attached. This information consists of an optional original node and an optional partial function mapping an error to a different type of error. Each error has a node attached to it which caused the error. When transforming an error the attached node is replaced recursively by the original node specified by the attached error transformation. Subsequently, if the partial function of the node is defined for the found error, it will be applied to the error to generate an error of the new type.

To display more informative error messages, each AST node has the position in the input file attached to it. When creating new nodes the position of the original node can be retained so the displayed error position is correct. For example, the generated inhales and exhales of a method call will have the position information of the original call statement. For most new nodes the original node is just the node which is currently being transformed and triggered the generation of this new node. In some cases this is not desirable as the output might be wrong. If, for example, the expression `-n` is rewritten as `-1 * n` and both nodes `-1` and `n` have `-n` as their original node, the back-translated expression will be `-n * -n`.

The partial function mapping error types is only used in a few places. Firstly, method contracts which are transformed into inhales and exhales have to be transformed from 'exhale failed' to 'postcondition violated' for methods or 'precondition false' for method calls. If the assignment of an argument for a method call fails (e.g., due to missing permission) the resulting 'assignment failed' has to be translated to 'call failed'. The second case where the type of an error has to be changed are invariants of while loops. The occurring 'exhale failed' would be a 'loop invariant not established' or 'loop invariant not preserved', respectively. If not enough permission for the loop condition is present, an 'inhale failed' will be generated in the desugared form once the condition is assumed. This is transformed into a 'while failed' error.

4.5 Plugin System

To be able to implement the extension as a plugin the Viper framework had to be extended by a plugin system which allows to load extensions and allows them to modify the program in suitable ways. Plugins can now be added without changing anything in Viper. If the plugin is on the class path it can be loaded using the `--plugin PluginClassName` (where `PluginClassName` is the fully qualified name of the class containing the plugin) argument when starting Viper. If several plugins are specified they will modify the program in the order they were specified. Thus, the developed extension is composable and can be combined with other extensions.

A plugin consists of a class which implements the trait `viper.silver.plugin.SilverPlugin`. It can override several hooks as needed to modify the program in different stages of parsing and translation. Should the plugin encounter an error (e.g., if the user inputs an unsupported expression) it can call the method `reportError` with the encountered error. As soon as any plugin reports any error the translation is terminated and verification is not performed anymore. The errors can still be processed by plugins but all other steps are omitted.

The plugin can modify the program/result at different stages by overriding the following methods (in order of execution):

beforeParse is called with the input string read from the input file before parsing started.

beforeResolve is called before identifiers (such as method calls) are resolved.

beforeTranslate is called before the parse AST is translated into the Viper AST.

beforeMethodFilter is called before the methods which should not be verified are removed.

beforeVerify is the last hook before the actual verification happens. Here the fully translated Viper AST will be available for rewriting.

mapVerificationResult allows to change the results of the verification. Error back translation should happen here.

beforeFinish is called just before the results are printed.

The ARP plugin developed for this project uses three of these hooks. During `beforeResolve` all instances of `rd` are rewritten to `rd()`. Using `rd` without parentheses is not possible, because at the positions it is used no such variable exists. A function call to a global function on the other hand can be resolved. Further, functions are added for all added syntax (e.g., `rd()`, `rdc(n)`, `rd_token(tk)`). This allows the resolving stage to find matching functions. The transformation of the program (i.e., desugaring, log-keeping and constraining) is performed in `beforeVerify`. Finally, in `mapVerificationResult` the found errors are translated back to the original program.

4.6 Nagini

Nagini translates the Python program directly into a Viper AST without generating a human readable text form. For this a Java virtual machine instance is started which loads the desired back-end for verification. The generated AST can then be passed directly to the verifier. Due to this, the implemented plugin system does not work as the hooks are not executed. Instead, the ARP plugin is manually loaded and the needed hooks (second phase and error back translation) are directly called.

Nagini provides a contract library to specify user specification in Python programs. To make ARP available a few new contracts were added. `ARP()` is translated to `rd()` in Viper. `Rd(x.f)` is a short form for `Acc(x.f, ARP())`. `ARP(n)` representing counting permissions gets translated to `rdc(n)` for an integer `n`. And finally `RD_PRED` is a constant representing the read permission used in predicates and gets translated to `globalRd()`.

Starting and joining a thread in Nagini are encoded as explicit inhales and exhales. When a new thread is started, the ARP used in the contract will be translated to `rd_token_fresh(tk)` where `tk` is the thread which was started. When joining a thread, ARP in the contract will be translated to `rd_token(tk)`. As the same thread object is used as a token, the same value which was used in the method which started the thread can be used to join the thread. To access the read permission used in a forked method, `GET_ARP(thread)` is used which is translated to `rd_token(thread)`.

Previously, Nagini only supported to use `0`, `1` or a fraction `a/b` (for integers `a` and `b`) in access predicates. To be able to encode more examples and use all of the added features the Viper extension provides this was extended to support arbitrary permission expressions (e.g., `1 - n * ARP()`).

4.7 Limitations

The implementation of the presented encoding which was done during this project has some limitations of what can and cannot be used in combination with ARP.

All syntax added for ARP may only be used in linear expressions. Otherwise it is not possible to correctly log the permissions and constrain the ARP. Furthermore, if ARP are used, ternary expressions may only occur as the top level expression in the permission part of an access predicate. These expressions can always be rewritten to a form where the ternary expression is outside the access predicate.

It is not allowed to assign any ARP to a variable. Otherwise, it would not be possible to accurately log the permissions. This is enforced using **`assert`** statements before assignments where the right hand side is a **`perm`** expression.

The built-in **`wildcard`** expression must only be used in functions. In all other places, the replacement `rdw` has to be used in order for the log to accurately reflect the held permission. **`wildcard`** and `rdw` are otherwise semantically equivalent.

Quantified access predicates are only supported if the accessed location has the form `x.f` for a bound variable `x` and a field `f`. If the reference part does not directly come from a local variable (e.g., a function result or another location access) our implementation is not able to correctly log the permissions. The condition as well as the permission expression can be any expression which is supported in quantifiers. The permission expression can depend on the quantified reference.

Packaging magic wands is not supported and if used in combination with ARP in the same method may lead to undefined behavior. This is due to missing log updates when packaging a magic wand. There is no point before or after the package statement where all permissions to the right hand side of the magic wand are guaranteed to be present. These permissions may only be attainable by for example unfolding predicates. These actions necessary are described in the proof script but it is not possible to do the logging in the body of the proof. This is caused by the fact that statements in the proof script have no effect outside the proof.

If the simple ARP encoding as described in Section 3.2.1 can be used, the above limitations do not apply. Notably, quantified expressions containing ARP can have any form.

Chapter 5

Evaluation

The encoding we presented in Section 3.2 allows to verify many interesting examples using ARP. ARP are not only available in Viper but are also implemented in Nagini. We managed to implement the extended Viper language without adding ARP specific code to the existing implementation by using the newly implemented plugin system. The extended Viper language provides intuitive new syntax to use ARP similar to the one used in Chalice. Section 5.1 shows some examples using ARP which can now be verified. Section 5.2 describes a new lightweight parser to improve performance measurements. In Section 5.3 we present the measured performance of the encoding with and without using ARP.

5.1 Expressiveness

Listing 28 shows the encoding of our initial example presented in Listing 2. In order to have access to the two subexpressions a recursive predicate is used. ARP are not only used in the method contract of the recursive method but also in the predicate.

Nagini now supports to use ARP in combination with threads as well as locks. We translated many examples which use ARP from the Chalice2Viper [6] test suite to Python to be verified using Nagini. The 24 examples can be found in the Nagini repository[1]. They test basic usage of ARP, permission arithmetic involving ARP as well as simple examples of forking and joining threads which use ARP. All translated examples were successfully verified with the VCG back-end. The SE back-end was able to verify almost all examples as well but three of the examples could not be verified.

One reason for the incompleteness of the SE back-end seems to be a constraint needed if ARP are used in some expressions in predicates. The constraint

[1]https://github.com/marcoeilers/nagini/tree/threads_with_arp/tests/arp/verification

```python
class Expr:

    def __init__(self) -> None:
        self.left = Expr()   # type: Expr
        self.right = Expr()  # type: Expr
        Ensures(Acc(self.left))
        Ensures(Acc(self.right))

    @Predicate
    def valid(self) -> bool:
        return Rd(self.left) and Rd(self.right) and \
               Implies(self.left is not None, self.left.valid()) and \
               Implies(self.right is not None, self.right.valid())

    def eval(self, state: State) -> int:
        Requires(self.valid())
        Requires(Rd(state.mapping))
        Ensures(self.valid())
        Ensures(Rd(state.mapping))
        Unfold(self.valid())
        result = 0  # type: int
        if self.left is not None:
            result += self.left.eval(state)
        if self.right is not None:
            result += self.right.eval(state)
        Fold(self.valid())
        return result
```

Listing 28: Initial example from Listing 2 encoded in Python. Nagini provides contracts to specify the program specification.

`none < write - n * globalRd()` is nonlinear due to the multiplication of the two constants `n` and `globalRd()`. Nonlinear constraints are known to cause incompleteness in SMT solvers. However, the examples could be verified in the VCG back-end.

The new encoding allows to verify examples which were not possible to verify in Chalice. We are able to encode the first example from the 2014 FTfJP paper by Boyland et al. [4] in Python and successfully verify it using the VCG back-end. Due to the issue mentioned above it is not possible to verify it in the SE back-end. The encoding of the first example can be found in Appendix A.1. It is not possible to verify the example in Chalice because ARP in negative positions will not be constrained. We were not able to verify the second example. To be able to verify this example it is necessary to transfer information about read permissions used to fork a method in one loop into an other loop where the method is joined again. While this is possible to encode for a single thread the example forks and joins an unbounded number of threads. One approach is to introduce a construct to express the read permission of a list of tokens [1]. This is currently not supported by the described encoding.

The new encoding also supports ARP in quantified permissions. It is possible to verify examples using quantified permissions if the location access is of the form `x.f` for a bound variable `x` (see Section 3.2.4 for details). Listing 29 shows a simple example using quantified read permissions we are able to successfully verify. The expression `Forall(self.seq, lambda r: Rd(r.x))` in Nagini is equivalent to the contract `forall r: Ref :: r in self.seq ==> acc(r.x, rd)` in Viper. The argument `b` is used to provide a base case in the recursion.

```python
def m1(self, b: bool) -> None:
  Requires(Rd(self.seq) and Forall(self.seq, lambda r: Rd(r.x)))
  Ensures(Rd(self.seq) and Forall(self.seq, lambda r: Rd(r.x)))
  if b:
    self.m1(not b)
```

Listing 29: Nagini example using quantified read permission in a recursive method.

5.2 Custom Parser for Performance Evaluation

While measuring performance of the produced encoding we encountered large differences between generating the encoding on the fly using the plugin and loading the generated encoding from a file. The second approach added a significant overhead for parsing and type checking the larger input. Viper Runner [8], which was used to measure performance, measures the duration between start of the program until termination. Due to the large overhead the results did not reflect the true performance differences. To get meaningful performance measurements the parse speed should be as fast as possible and should be the same for all inputs.

In order to eliminate these differences we implemented a second approach to load a program for verification which allows to bypass parsing and type checking. In a first step the AST of the program to verify is loaded using the normal procedure used in Viper. This AST is then written to a file using a simple to parse format. During performance testing this file is read, the AST is reconstructed and it can directly be passed to the verification stage. Due to the simple structure of the used format parsing is easy and can be done with one single pass over the file. Type checking can be omitted as the AST is known to be well-formed and everything is already resolved. Using this new parsing method removed the overhead and resulted in similar measurements as when generating the encoding on the fly.

The syntax used to store the AST is similar to Lisp. An instance of the case class classname is represented as `(classname|param_1|param_2|...|param_n)` where `param_n` is the nth parameter of the case class. The parameters are encoded recursively using the same format. To load a program the whole string can be read once from the start. Upon encountering an opening parenthesis the first part can be read to learn the case class. It is then known how many arguments are expected. The arguments can be read recursively until all arguments were parsed. The case class can then be instantiated. As all case classes are dumped with all their parameters it is never necessary to search for something in another part of the program. The improved parsing speed comes with a larger file size and it is no longer possible to manually edit the program.

This was implemented using the plugin system described in Section 4.5. Exporting is done in `beforeVerify` where the finished AST is available and can be dumped to file. When loading a file the input is parsed in `beforeParse` before Viper sees the input string and is stored locally. An empty program is returned to be 'parsed' by Viper. In `beforeVerify` the previously stored AST is returned and can be verified.

5.3 Performance

We measured the performance of the encoding for different scenarios. The scenarios are compared between the usage of ARP and using concrete fractions instead. Where possible, the difference between the log-based encoding and the simple encoding are compared. We created some examples in Viper which use one specific language feature to measure the impact of the encoding on these features.

All measurements were performed on a Dell Latitude E7440 with an Intel Core i7-4600U CPU @ 3.3GHz and 8 GB of RAM running a fresh install of Ubuntu 16.04 Xenial. The measurements were performed using Viper Runner [8], each configuration was measured 20 times. The times mentioned below are averages over those 20 runs. All examples were tested with the SE and VCG back-ends. To be able to reliably measure verification time and not overhead caused by parsing and typechecking, a custom parser as described in Section 5.2 was used for most measurements.

The examples which were used can be found in the repository[2].

5.3.1 Examples without ARP

We measured the verification time of Viper examples which do not use ARP but were encoded using the log-based encoding. We compared this time with the verification time for the original input. This gives an impression how expensive desugaring and log-keeping is. Figure 5.1 shows the measured times and slowdown. We see a slowdown of a factor of 1.4 to 2.4 for the SE back-end and a slowdown of 1.2 to 2 for the VCG back-end.

Back-end	**Example**	**Normal [s]**	**Encoded [s]**	**Slowdown**
SE	Fields	0.31 (0.08)	0.51 (0.14)	1.62
	Predicates1	0.29 (0.08)	0.48 (0.13)	1.63
	Predicates2	0.32 (0.10)	0.77 (0.20)	2.39
	Quantifiers	0.76 (0.12)	1.74 (0.26)	2.29
	Test Suite	0.26 (0.15)	0.48 (0.52)	1.85
VCG	Fields	1.57 (0.07)	3.19 (0.25)	2.04
	Predicates1	2.54 (0.06)	3.16 (0.13)	1.24
	Predicates2	1.66 (0.07)	3.36 (0.20)	2.02
	Quantifiers	1.61 (0.07)	3.23 (0.13)	2.01
	Test Suite	2.47 (1.59)	3.31 (1.53)	1.34

Figure 5.1: Performance of examples without ARP. The number in parentheses is the standard deviation.

As the examples use one specific feature a lot and do not represent an average usage pattern, the performance on some examples from the Viper test suite was measured as well. The examples measured are the ones located in `all/basic` in the test suite. The SE back-end had an average slowdown of 1.67 while the VCG back-end had an average slowdown of 1.75. The table in Figure 5.1 shows the average runtime of the examples from the test suite and the slowdown based on those averages.

[2]https://bitbucket.org/viperproject/arp-plugin/src/0f409f7da455889175f92967b4ac9e3827e707e6/src/test/resources/arp/performance/

To find out which part of the encoding is responsible for the increased verification time we deleted parts of the generated encoding to see which part decreases the verification time the most.

In a first step the dummy methods used for contract well-formedness checks were removed. Secondly, the log-keeping was removed. As no ARP was used, the program behavior was not changed. Then, the domain used for log-keeping as well as the domain providing unique field identifiers was deleted. In a last step, all unnecessary remains were removed. This includes for example the additional argument to pass the read permission to be used in contracts.

These measurements were performed for all performance examples with both back-ends. Figure 5.2 shows the measured times and slowdown relative to the unencoded form for the example 'fields' with the SE back-end. Most parts account for some slowdown but there is no huge slowdown in any of the parts measured. The largest difference seems to be the log-keeping. The results were similar for most of the cases.

File	**Time [s]**	**Slowdown**
Fields	0.36 (0.11)	1
Encoded full	0.59 (0.17)	1.64
Encoded without dummy methods	0.51 (0.15)	1.4
Encoded without log-keeping	0.41 (0.12)	1.14
Encoded without domain	0.37 (0.10)	1.03
Encoded minimum	0.36 (0.10)	1.01

Figure 5.2: Performance of examples where part of the encoding was removed. Starting from 'Encoded full' more and more parts of the encoding were removed. The largest drop occurs after removing the log updates. The number in parentheses is the standard deviation.

Using many predicates with parameters results in a quadratic number of axioms to ensure uniqueness of the corresponding identifiers. We first suspected those axioms to be the cause of the large slowdown but the measurements did not confirm this suspicion. Removing the axioms had almost no influence on the performance and the largest drop occurred when removing the log updates as in all other examples.

5.3.2 Examples with ARP

The performance impact of actually using ARP largely depends on the type of usage. We used the same Viper examples as for the measurements without ARP, but replaced concrete values by an ARP. In the examples the measured slowdown ranged from 1.02 to 44.32. The simple encoding was measured as well as the log-based encoding. Figure 5.3 shows measurements when using ARP. Slowdown is calculated relative to the encoded form and not the original input. This is to show the additional impact of actually using ARP in the new encoding.

Entries marked with * did not successfully verify with the default settings. By increasing the timeout for SMT queries in the SE back-end we were able to verify the examples. Even though the same timeout was used for all settings of those examples the slowdown is much larger than for the other examples. The verification times also varied a lot for those examples. For example, the SE back-end took between 1.02 and 40.81 seconds to verify the encoded form of the ‘predicates2‘ example. Very likely, this large difference between the two back-ends and the large slowdown in the SE back-end is due to branching. There are many branches needed to check the log and constrain read permissions. While the VCG back-end is able to handle branches quite well, the SE back-end experiences exponential performance degradation if many branches are used.

In general the performance impact of using the encoding is noticeable and there is some room to enhance the performance in the future. One possible approach is to reduce the number of needed branches. The VCG back-end seems to be better suited for the current encoding as it shows for many examples a rather low slowdown.

5.3.3 Nagini

We also measured the performance impact for Python examples in Nagini. As the additional contracts are translated directly into the extended Viper language, we expect a slowdown similar to the one measured in Viper. Table 5.4 shows measured run times for some Python examples from the Nagini test suite. As expected the measurements are very similar to the ones measured directly in Viper. We see on average a total slowdown between not encoded examples without ARP and the same examples using ARP of around 1.55 for the SE back-end and 1.53 for the VCG back-end.

Back-end	**Example**	**Encoded w/o ARP [s]**	**Simple w/ ARP [s]**	**Simple slowdown**	**Log-based w/ ARP [s]**	**Log-based slowdown**
SE	Fields	0.51 (0.14)	0.60 (0.13)	1.17	22.04 (7.91) *	43.30
	Predicates1	0.48 (0.13)	0.38 (0.10)	0.79	1.13 (0.16)	2.37
	Predicates2	0.77 (0.20)	4.49 (0.26)	5.81	17.45 (13.20) *	22.59
	Quantifiers	1.74 (0.26)	39.16 (2.78)	22.48	77.21 (1.43)	44.32
VCG	Fields	3.19 (0.25)	3.81 (0.24)	1.20	7.19 (0.13)	2.26
	Predicates1	3.16 (0.13)	3.82 (0.25)	1.21	3.23 (0.22)	1.02
	Predicates2	3.36 (0.20)	4.27 (0.22)	1.27	38.20 (4.14)	11.38
	Quantifiers	3.23 (0.13)	3.89 (0.24)	1.20	3.75 (0.13)	1.16

Figure 5.3: Examples where ARP are used. Entries marked with '*' did not successfully verify with the default settings and we increased the SMT timeout. The number in parentheses is the standard deviation.

Back-end	Example	**Not encoded [s]**	**Encoded w/o ARP [s]**	**Slowdown w/o ARP**	**Encoded w/ ARP [s]**	**Slowdown w/ ARP**
SE	Call	1.84 (0.13)	2.63 (0.20)	1.43	3.40 (0.21)	1.29
	Thread	2.23 (0.14)	3.02 (0.23)	1.36	3.40 (0.23)	1.13
	Simple	1.86 (0.13)	2.42 (0.18)	1.30	2.46 (0.23)	1.02
	Predicate	1.86 (0.14)	2.56 (0.20)	1.37	2.62 (0.23)	1.02
VCG	Call	2.82 (0.10)	3.80 (0.17)	1.35	6.05 (0.21)	1.59
	Thread	3.28 (0.10)	4.36 (0.13)	1.33	4.53 (0.18)	1.04
	Simple	2.81 (0.08)	3.67 (0.13)	1.31	3.71 (0.14)	1.01
	Predicate	2.87 (0.08)	3.80 (0.15)	1.32	3.86 (0.20)	1.02

Figure 5.4: Performance of examples in Nagini. The number in parentheses is the standard deviation. Slowdown is relative to the previous column (i.e., slowdown with ARP is relative to encoded form without ARP).

Chapter 6

Conclusion

In this project we implemented a new encoding for abstract read permissions. While ARP are implemented in Chalice the new encoding is more powerful and allows to verify more examples. In particular, it allows to verify the first example presented in the 2014 FTfJP paper by Boyland et al. [4].

The new encoding was implemented as an extension to the existing Viper verification infrastructure. As part of this, a plugin system for Viper was implemented. The used Viper-to-Viper translation is composable and allows to add more extensions at the same time. The plugin supports not only the cases defined in Section 1.1 (ARP, counting permission, wildcards) but also token permissions, which were used to support threads in Nagini. Therefore, all types of ARP supported by Chalice are supported by the extended Viper language. Errors can be successfully translated back to the original program.

ARP in quantified expressions are supported if the simple encoding (see Section 3.2.1) can be used or some constraints are satisfied (see Section 3.2.4). We successfully verified many examples using quantified permissions in Viper as well as in Nagini.

Nagini has been extended to support usage of the features provided by the developed extension. The newly added syntax is similar to the already existing Nagini syntax.

Finally, we evaluated our implementation using examples from the Chalice2Viper test suite as well as the first example from the paper introducing the constraint system we used [4]. While the performance impact is not negligible it is still usable especially with the VCG back-end.

Future Work. While ARP can now be used in Viper and in front-ends, there is still room for future work. The performance of the encoding can be improved by optimizations in the log-keeping, constraining or desugaring of the program.

For example, one could investigate how to reduce the number of branches used in the encoding since it significantly affects performance in the SE back-end.

The second example from the constraint paper [4] was not yet encoded with the new ARP encoding. Adding new constructs to express read permissions of a sequence of tokens would allow to encode the example.

Moreover, while quantified permissions are supported in many cases this can be further expanded to lift the limits mentioned in Section 4.7. One could reuse ideas used in the implementation of quantified permissions in the SE back-end [13] to extend the current implementation to the general case. By finding suitable inverse functions for the locations in the quantified access predicate, it might be possible to correctly log these permission changes.

Packaging magic wands is currently not supported because the permissions cannot always be logged. This could probably be supported by implementing ARP directly in the back-end.

Finally, it would be interesting to integrate ARP in permission inference to directly generate a read permission.

Bibliography

[1] Vytautas Astrauskas. Encoding of chalice permissions in viper. unpublished, 2017.

[2] Stefan Blom and Marieke Huisman. The vercors tool for verification of concurrent programs. *FM*, 8442:127–131, 2014.

[3] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *ACM SIGPLAN Notices*, volume 40, pages 259–270. ACM, 2005.

[4] John Tang Boyland, Peter Müller, Malte Schwerhoff, and Alexander J Summers. Constraint semantics for abstract read permissions. In *Proceedings of 16th Workshop on Formal Techniques for Java-like Programs*, pages 1–6. ACM, 2014.

[5] ETH Zurich Department of Computer Science. Encoding ADTs. `http://viper.ethz.ch/examples/encoding-adts.html`, 2016. [Online; accessed 2018-02-02].

[6] ETH Zurich Department of Computer Science. Chalice2Viper. `https://bitbucket.org/viperproject/chalice2silver`, 2018. [Online; accessed 2018-02-02].

[7] ETH Zurich Department of Computer Science. Nagini. `https://github.com/marcoeilers/nagini`, 2018. [Online; accessed 2018-02-12].

[8] ETH Zurich Department of Computer Science. Viper Runner. `https://bitbucket.org/viperproject/viper-runner`, 2018. [Online; accessed 2018-02-02].

[9] Simon Fritsche. *A Framework for Bidirectional Program Transformations*. Master's thesis, ETH Zurich, 2017.

[10] Stefan Heule, K Rustan M Leino, Peter Müller, and Alexander J Summers. Abstract read permissions: Fractional permissions without the fractions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 315–334. Springer, 2013.

[11] K Rustan M Leino and Peter Müller. A basis for verifying multi-threaded programs. In *ESOP*, volume 9, pages 378–393. Springer, 2009.

[12] 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.

[13] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In *International Conference on Computer Aided Verification*, pages 405–425. Springer, 2016.

[14] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. In *Proceedings of the 10th ECOOP Workshop on Formal Techniques for Java-like Programs*, pages 1–12, 2008.

Appendix A

Appendix

A.1 First Example from Constraint Paper

The encoding of the first example from the 2014 FTfJP paper by Boyland et al. [4] in Python.

```python
from nagini_contracts.lock import Lock
from nagini_contracts.contracts import *
from nagini_contracts.obligations import Level, WaitLevel, MustTerminate
from nagini_contracts.thread import Thread


class Cell:
    def __init__(self, val: int) -> None:
        self.value = val
        self.rds = 0
        Ensures(Acc(self.value) and self.value == val)
        Ensures(Acc(self.rds) and self.rds == 0)


class CellLock(Lock[Cell]):

    @Predicate
    def invariant(self) -> bool:
        return Acc(self.get_locked().rds) and self.get_locked().rds >= 0 and \
               Acc(self.get_locked().value, 1 - self.get_locked().rds * ARP())


class Writer:
    def write(self, data: Cell) -> None:
        Requires(Acc(data.value))
        Requires(MustTerminate(2))
        Ensures(Acc(data.value))


class Reader:
    def read(self, data: Cell) -> None:
        Requires(Rd(data.value))
        Requires(MustTerminate(2))
        Ensures(Rd(data.value))


class RWController:
    def __init__(self, c: Cell) -> None:
        Requires(Acc(c.rds) and Acc(c.value) and c.rds == 0)
        Ensures(Acc(self.c) and self.c == c and Acc(self.lock) and \
                self.lock.get_locked() is self.c)
        Ensures(WaitLevel() < Level(self.lock))
        self.c = c  # type: Cell
        self.lock = CellLock(self.c)  # type: CellLock

    def do_write(self, writer: Writer) -> None:
        Requires(writer is not None)
        Requires(Rd(self.lock) and Rd(self.c) and \
                 self.lock.get_locked() is self.c)
        Requires(WaitLevel() < Level(self.lock))
        Ensures(Rd(self.lock) and Rd(self.c))
        self.lock.acquire()
        Unfold(self.lock.invariant())
        if self.c.rds != 0:
            Fold(self.lock.invariant())
            self.lock.release()
            self.do_write(writer)  # try again
        else:
            writer.write(self.c)   # lock acquired successfully
            Fold(self.lock.invariant())
            self.lock.release()

    def do_read(self, reader: Reader) -> None:
        Requires(reader is not None)
        Requires(Rd(self.lock) and Rd(self.c) and \
                 self.lock.get_locked() is self.c)
        Requires(WaitLevel() < Level(self.lock))
        Ensures(Rd(self.lock) and Rd(self.c))
        self.lock.acquire()
        Unfold(self.lock.invariant())
        self.c.rds += 1
        Fold(self.lock.invariant())
        self.lock.release()
        reader.read(self.c)
        self.lock.acquire()
        Unfold(self.lock.invariant())
        self.c.rds -= 1
        Fold(self.lock.invariant())
        self.lock.release()
```

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Abstract Read Permission Support for an Automatic Python Verifier

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Schmid	Benjamin

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Sarnen, 03.03.2018	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.