Extending Viper with user-defined permission models

Master's Thesis Project Description

Matthias Roshardt
Supervisors: Thibault Dardinier, Prof. Peter Müller

March 2021

1 Introduction

Computer program behaviour deviating from what the programmer expects is a fundamental problem in software engineering with potentially severe consequences. The traditional approach of probing programs with test cases is error-prone, or as Edsger Dijkstra put it: "Testing can be used to show the presence of bugs, but never show their absence". This is what makes formal verification techniques, which can guarantee desirable properties about a program *for all possible executions*, a promising option. Particularly for heap-manipulating programs and thread interactions in concurrent software, this is a challenging task. Based on earlier theoretical work, automated verifiers such as Viper [1] have been created to make formal verification more accessible by providing the user with strong automation for advanced language features. Viper in particular, which this thesis revolves around, aims at providing a common abstraction layer that can be built on to create verifiers for any choice of programming language (e.g. Nagini [2] for Python, Prusti [3] for Rust, VerCors [4] for OpenCL).

When verifying the correctness of a large computer program, a modular approach on the level of individual functions and methods is needed, both for reasons of time complexity and accessibility. In the end, we would like to be able to be able to state pre- and postconditions for a method, as well as make assertions about the program state within the method. This requires reasoning about the values of local variables but also of values on the heap. The latter introduces a threefold problem. One, the verifier needs to able to reason about the method in question without having to consider every possible heap configuration that has been created by the program at large. Two, it needs to be able to verify whether a method call modifies any of the heap values accessed in the current method. Three, in a concurrent setting, it needs to be able to reason about whether another thread might have modified a heap value accessed in the current method.

1.1 Separation Logic

To address the above requirements, *separation logic* [5] was formulated. It is in essence an extension of Hoare Logic, where in addition to statements about variables in the store we can make statements about the heap. The heap is usually defined as a partial map from the set of addresses to the set of values. The simplest heap is the empty one, i.e. the one where the map is undefined for all addresses. The next simplest type of heap has a single value stored at a particular address. Such a heap can be described with the following separation logic assertion: $a \mapsto v$, which is to say "at address a on the heap, the value v is stored", or simply "a points to v". Heaps can be combined if their (address) domains are disjoint. Given two assertions about heaps P, Q (which might be points-to assertions as seen above), the assertion $P * Q$ denotes the statement "There exist two heaps h_1, h_2 such that this heap can be created by combining h_1 with h_2, P holds for h_1 and Q holds for h_2". This gives us all the language we need to reason about heaps. On the calculus side, the central addition of separation logic is the the so-called *frame rule*,

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}}\mathsf{mod}(C) \cap \mathsf{fv}(R) = \emptyset$$

which, put simply, states that if a program C executes safely in a small state [1], then it also executes safely in a larger state, and moreover, nothing beyond the smaller state is affected by the program. It is this rule that allows for local reasoning about programs - the local context can be logically isolated from the broader one.

1.2 Permissions

What a verifier still needs to actually succeed at proving e.g. that a heap value is still the same after a function call, is a notion of ownership. At a basic level, we want to encode the semantics that a program location may only perform operations (such as reading or writing) on a heap value if it owns that heap value, or in other words, has *permission* to perform that operation on it. The points-to assertion we saw before already provides us with a basic notion of ownership, as in: if a program section has $a \mapsto v$ as a precondition, then it can be said to own this heap value and is free to read and modify it.

To enable more sophisticated permissions than all-or-nothing, we extend our notion of state, such that the heap is now not just a map from addresses to values, but instead a map from addresses to a value paired with a *share*[2] (which encodes the permission).[3] Our points-to statements now take the form $a \mapsto (s, v)$

[1] A state consists of a *store*, i.e. a map of variables to values, and a *heap*, i.e. a partial map of addresses to values

[2] As in, a share of a resource (in this case the value associated with the address), which is a phrase that lends itself well to a natural interpretation that one can either own the resource completely, only some parts of it, or none at all.

[3] Of course, this requires extending our earlier notion of "joining" states required e.g. by the separating conjunction ($*$) operator of separation logic, such that shares can be joined in a

(more commonly denoted $a \mapsto_s v$), which is to say "I have s permission to the value v stored at address a". Let's have a look at what x might be.

1.2.1 Boolean shares

The most straightforward option is to simply give two permissions: no permission at all and full permission. Obviously we must ensure that it is impossible for two or more locations to hold a full permission to the same reference. Along that same vein, we must have a way to specify which program section is supposed to hold which permission.

In Viper, this can be expressed as part of method preconditions (keyword `requires`) and postconditions (keyword `ensures`). As can be seen in listing 1, the `increment` method requires permission to access field `f` before it can read from or write to it. At the end, we must ensure that the method still holds the permission, so that it can be "given back" to the client method. If this were omitted, the verifier could not prove that the `client` method can reacquire the necessary permission to subsequently read from the field, the permission could have been *leaked*. Analogously, the `client` method requires a full permission to `f`, because otherwise it might not be possible to satisfy `increment`'s precondition.

In this example, the permission transfer is implicit through the use of pre- and postconditions. Viper also provides the keywords `inhale` and `exhale` which explicitly express that a piece of code (within a function) acquires or relinquishes the specified permissions.

```
field f: Int

method client(a: Ref)
        requires acc(a.f)
        ensures acc(x.f) {
        increment(a)
        var b : Int
        b := a.f
}

method increment(x: Ref)
        requires acc(x.f)
        ensures acc(x.f) {
        x.f := x.f + 1
}
```

Listing 1: Example use of permissions in Viper

1.2.2 Fractional shares and beyond

The Boolean share model presented above is too restrictive; many useful concepts cannot be expressed, such as (concurrent) read-only access. We could envision a

way that preserves their semantics.

ternary share model, with an empty permission, readonly permission and full permission. However, we quickly run into trouble when defining the semantics of how to join such permissions together.[4] Instead, Viper uses *fractional permissions*, where a share is a rational number between 0 and 1. Write access requires a share of 1, whereas read access requires any share greater than 0. The benefit of this permission model is that it naturally accommodates concurrent programming: a thread with some permission may split this permission into fractions, fork a number of child threads (which might have children of their own) that perform some sort of computation requiring read access, and recombines the shares as the child threads join again.

However, this permission model also has some weaknesses: on one hand, the fact that there can be many identical shares (e.g. two instances of 0.1) leads to some unexpected behaviour (for instance, if one naively defines trees, one will end up defining DAGs instead [7]). On the other hand, it is impossible to know how many shares are out there, as any nonzero share can be split up indefinitely. A number of alternative models have been proposed [6], each with their own benefits and drawbacks. The aim of this master's thesis is to overcome these limitations in Viper by implementing one or more additional advanced permission models and ultimately by letting the user define custom models that suit his or her purposes.

2 Core Goals

2.1 Exploration

The first step in this thesis is to get a good picture of the design space of permission models. The goal is to include a comprehensive overview of the landscape of permission models in the thesis. Limitations of the different models, particularly of fractional permissions, are illustrated with examples. Alternative memory models (different from the standard address $\rightarrow$ value map) may also be explored.

2.2 Automation

Before making the step to fully parametrized models, at least one advanced permission model (such as tree shares as presented in [6]) is implemented in one of Viper's two back-end verifiers. An evaluation is performed vis-à-vis fractional permissions.

2.3 User-parametrized models

Viper is extended such that users can encode their own permission models. What the specification looks like is subject to exploration. A specification may be comprised of:

[4] The explanation for this is beyond the scope of this document.

- the set of shares,
- the algebraic structure of the join operation on them (i.e. what happens when two permissions are added, multiple permissions are inhaled etc),
- which shares correspond to a read/write/zero permission and
- the product operator for predicates. [5]

The soundness of the user-defined permission model would preferably be verified. If possible, the implementation should check the necessary properties automatically. Alternatively, the user can be charged with proving them, if the former approach is not feasible or for other reasons less desirable. A hybrid approach may yield the best results.

2.4 Evaluation

Encode various permission models from the literature using the above extension and evaluate them versus the previously added advanced model and default (fractional) model with regards to performance and reliability.

3 Extension Goals

3.1 User-parametrized automation

If the hard-coded model from the automation goal outperforms the models encoded in the general interface w.r.t. speed and reliability, the thesis explores how the user can aid the automation carried out by the verifier by specifying additional properties of the model.

3.2 Properties and soundness

Built into the semantics of Viper is the assumption that several properties hold for the permission model (e.g. holding a full (write) permission implies that no other thread can hold a non-zero (read) permission simultaneously). The goal is to identify what these properties are and evaluate their necessity, providing examples for unexpected behavior in case they are violated. Lastly, the thesis explores which properties could be axiomatized and which ones could be left to the user to prove.

3.3 Advanced features

Beyond the basic support for parametrized permission models as outlined above, there are a number of advanced features that can be adapted to work with the new permission models. The goal is to extend the work done in the thesis to such things as:

[5] In Viper, fractional amounts of (recursive) predicates can be folded or unfolded, which requires a multiplication operation.

- Known-folded permissions
- Permission introspection
- Magic wand operator
- Permission wildcards

References

[1] P. Müller, M. Schwerhoff and A. J. Summers. *Viper: A verification infrastructure for permission-based reasoning.* In: Verification, Model Checking, and Abstract Interpretation. VMCAI 2016. Lecture Notes in Computer Science, vol 9583. Springer, Berlin, Heidelberg.

[2] M. Eilers and P. Müller. *Nagini: A static verifier for Python.* In: Computer Aided Verification (CAV). CAV 2018. Lecture Notes in Computer Science, vol 10981. Springer, Cham.

[3] V. Austrauskas, P. Müller, F. Poli and A. J. Summers. *Leveraging Rust Types for Modular Specification and Verification.* Proc. ACM Program. Lang. 3, OOPSLA, Article 147, October 2019.

[4] S. Blom and M. Huisman. *The VerCors Tool for Verification of Concurrent Programs.* In: Formal Methods. FM 2014. Lecture Notes in Computer Science, vol 8442. Springer, Cham.

[5] J. C. Reynolds. *Separation Logic: A logic for shared mutable data structures.* Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, Copenhagen, Denmark, 2002, pp. 55-74.

[6] R. Dockins, A. Hobor and A. W. Appel. *A fresh look at separation algebras and share accounting.* In: Programming Languages and Systems. APLAS 2009. Lecture Notes in Computer Science, vol 5904. Springer, Berlin, Heidelberg.

[7] X. Le, T. Nguyen, W. Chin and A. Hobor. *A certified decision procedure for tree shares.* In: Formal Methods and Software Engineering. ICFEM 2017. Lecture Notes in Computer Science, vol 10610. Springer, Cham.