# A General Approach to Formally Verify Viper Front-ends

Master Thesis - Project Description
Supervised by Prof. Dr. Peter Müller, Thibault Dardinier and Gaurav Parthasarathy

Hongyi Ling

November 9, 2023

## 1 Motivation

Program verification techniques enable proving the absence of bugs in programs, which is becoming increasingly important given the widespread use of programs in critical domains. Separation logic (SL) [9] is one such program verification technique for heap-manipulating programs. It provides a way to prove Hoare triples $\{P\}\ c\ \{Q\}$, where $P$ is a precondition, $c$ is the code to be executed and $Q$ is a postcondition. A Hoare triple $\{P\}\ c\ \{Q\}$ holds if whenever $P$ holds before the execution, then $Q$ holds after the execution of $c$. Concurrent separation logic (CSL) [8] further extends SL to verify concurrent programs. It enables $c$ in the Hoare triple $\{P\}\ c\ \{Q\}$ to contain concurrency, and asserts the triple only if it holds for all possible interleavings in parallel executions of $c$.

In practice, it is desirable to automatically verify source code statically. Viper [7] is an automatic verification infrastructure, which includes the Viper intermediate verification language (IVL) and verifiers to automatically check the correctness of Viper IVL programs. The assertion language in the Viper IVL is based on SL. The Viper infrastructure has been used as a foundation to develop various verifiers for real-world programming languages using SL and CSL reasoning principles. This is achieved by developing Viper *front-ends* that translate input programs and their specifications, such as pre- and postconditions, into Viper IVL programs. Examples for Viper front-ends include Gobra [11] for Go, Nagini [6] for Python, Prusti [2] for Rust, and VerCors [3] to verify Java Programs.

In order for such a front-end translation to be *sound*, the correctness of the Viper program must imply the correctness of the input program w.r.t. the input specification. In general, the soundness of translations in current Viper front-ends is not formally proved. In recent work, Arlt [1] started addressing this issue, by

formally defining a simple Viper front-end that supports concurrency and proving its translation sound once and for all in the Isabelle theorem prover. Arlt's work shows the feasibility of formally proving the soundness of Viper front-end translations, where the proof relies on the soundness of CSL proof rules for the developed front-end. However, the work was specific to a particular translation. Moreover, the work deals directly with the operational semantics of a Viper program. This requires proving low-level steps to bridge the gap between Viper's operational semantics and the front-ends' CSL, which is cumbersome.

Our project aims to explore a general approach for proving Viper front-end translations sound once and for all. Moreover, instead of directly working with the correctness of a Viper program via its operational semantics, we want to instead work with a separation logic proof representation of a Viper program (such a representation already exists).

# 2 New Approach

To show the soundness of the translation from a front-end program to a Viper program, we need to prove that if the Viper program is correct w.r.t. Viper's operational semantics, then the front-end program is also correct w.r.t. to the front-end's operational semantics. In the new approach, we will construct this proof in three steps:

1. First, if the Viper program is correct w.r.t. its operational semantics, then there exists a valid SL proof for it. The existence has already been proved formally.

2. Second, convert the SL proof of a Viper program into a CSL proof of the front-end program. Developing a general approach for dealing with this step is the main goal of this project.

3. The soundness of a front-ends' CSL guarantees that a CSL proof of a front-end program implies the correctness of the front-end program w.r.t. the front-ends' operational semantics. The focus of this work is not on proving the CSL soundness (but we could use the approach by Vafeiadis [10] to prove such a result).

Our hope is that constructing the proof by connecting an SL proof of the Viper program with a CSL proof of the front-end will lead to a proof that more directly matches the intuition for why the translation is sound compared to Arlt's approach [1], which directly dealt with Viper's operational semantics.

In addition to applying the new proof strategy, we also want the approach to work for different kinds of front-end languages and corresponding CSLs. This will require abstracting various components, including the state model and the *leaf statements* of the front-end (such as assignments and allocations, but not, for example, sequential or parallel compositions). Our goal is to provide generic building blocks (in the form of formally proved lemmas) that can be used to systematically construct the

proof of a translation. For example, we want to prove general rules for reasoning about the translation of compositional front-end constructs (e.g. parallel and loop compositions), where we will impose requirements on the leaf statements of the front-end. To ensure that our rules are useful, we will use them to prove a concrete translation.

One challenge is that our building blocks will have to account for nontrivial mismatches between the front-end state and the Viper state. For example, in Arlt's work [1], the front-end's state model distinguishes between *owning* a heap location or not owning the heap location, while the Viper semantics supports fractional ownership of heap locations [4] represented by rationals. Dealing with this gap is nontrivial, because Viper may be reasoning with ownership amounts that cannot be represented in the front-end state. While Arlt's solution is specific to this particular mismatch, we aim to develop a general approach. In particular, our approach should also be able to handle the case when a front-end uses fractional ownership represented by rationals (as is standard in the literature) and when the Viper semantics uses fractional ownership represented by reals (which is what the actual Viper verifiers use, since SMT solvers have built-in support for reals but not for rationals). Arlt's approach does not work for the mismatch between rationals and reals.

# 3 Project Goals

## 3.1 Core Goals

1. Develop a general framework mechanised in Isabelle for proving the soundness of concurrency-supporting Viper front-ends. The framework should support the following compositional front-end constructs: sequential composition, if statements, while loops, and parallel composition. The front-end leaf statements, the front-end state model, and the Viper state model should be parameters of the framework. The framework should provide sufficient conditions on the front-end and Viper state models to ensure the soundness of different parts of the front-end translation. These conditions should be sufficient to handle the following two instantiations of the state models: (1) the front-end state model distinguishes only between owning and not owning a heap location (as in Arlt's work) and the Viper state model supports fractional permissions represented by reals, (2) the front-end state model supports fractional permissions represented by rationals and the Viper state model supports fractional permissions represented by reals.

2. In Isabelle, formalise a concurrency-supporting front-end language (syntax and semantics) that supports heap operations and all of the compositional constructs supported by the framework defined in core goal 1. The front-end language must support allocating, writing, and reading heap locations.

   In Isabelle, define the CSL assertion language for the developed language.

The assertion language must support fractional permissions represented by rationals.

In Isabelle, define an executable translation from the front-end language and the annotations (such as pre- and postconditions expressed in the CSL assertion language) to Viper.

3. In Isabelle, instantiate the framework defined in core goal 1 for the language and translation defined in core goal 2. The instantiation should prove the soundness theorem of the translation defined in core goal 2 (assuming the soundness of the front-ends' CSL).

### 3.2 Extension Goals

1. In Isabelle, show that the general framework is able to deal with more advanced parallel constructs (e.g., conditional critical regions used by O'Hearn [8] and Brookes [5]). This may require extending the framework.

2. In Isabelle, extend the CSL assertion language with more advanced features (for example, recursive predicates) and use the framework to obtain the corresponding soundness theorem for the adjusted translation.

3. Document the strengths and weaknesses of the new approach developed in the core goals compared to Arlt's approach.

## References


[1] Ellen Arlt. A formally verified automatic verifier for concurrent programs. Master's thesis, ETH Zürich, May 2023.

[2] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. Leveraging Rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[3] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The VerCors tool set: verification of parallel and concurrent software. In *Integrated Formal Methods: 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings 13*, pages 102–110. Springer, 2017.

[4] John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003. doi: 10.1007/3-540-44898-5_4. URL https://doi.org/10.1007/3-540-44898-5_4.

[5] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007.

[6] Marco Eilers and Peter Müller. Nagini: a static verifier for Python. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I 30*, pages 596–603. Springer, 2018.

[7] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*, pages 41–62. Springer, 2016.

[8] Peter W O'Hearn. Resources, concurrency, and local reasoning. *Theoretical computer science*, 375(1):271–307, 2007.

[9] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[10] Viktor Vafeiadis. Concurrent separation logic and operational semantics. *Electronic Notes in Theoretical Computer Science*, 276:335–351, 2011.

[11] Felix A Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C Pereira, and Peter Müller. Gobra: Modular specification and verification of Go programs. In *International Conference on Computer Aided Verification*, pages 367–379. Springer, 2021.