ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Improving a Deductive Program Verifier for Hyperproperties

Bachelor's Thesis

Paul Winkler

April 24, 2025

Advisors: Anqi Li, Thibault Dardinier, Prof. Dr. Peter Müller
Department of Computer Science, ETH Zürich

Abstract

Hypra is a deductive program verifier for Hyper Hoare Logic (HHL). It is based on generating verification conditions for HHL, which is a powerful logic capable of proving or disproving arbitrary trace properties and hyperproperties. The latter are properties relating multiple executions of a program and can express various interesting properties, such as security properties. However, Hypra has certain limitations. It solely supports integer-typed variables, restricting supported programs, and its error messages are imprecise and often unhelpful.

In this thesis, we present various improvements to overcome these limitations. First, we add support for boolean-typed variables to Hypra and introduce three type constructors for Sets, Sequences, and Maps. This broadens the range of supported programs and properties.. We further extend Hypra's previous suite of benchmarks by adding newly supported properties and analyze their performance. Our evaluation shows that Hypra can support these while requiring an acceptable runtime and number of proof annotations, although both can degrade noticeably in some cases. We improve error messages by designing clear guidelines and incorporating them into Hypra. We also present a Visual Studio Code extension for Hypra, which supports common features like syntax highlighting and error-handling.

Contents

Chapter 1

Introduction

The first chapter provides an overview of our motivation and outlines the structure of this thesis.

Defined as design defects in an engineering system that cause unwanted behavior, bugs are a major problem in engineering disciplines. They can inhibit the development process and cause significant financial or intellectual damage. Their occurrence can be prevented by *formal verification*, the process of establishing mathematical guarantees about programs under certain specifications [5]. It allows us to prove the absence of bugs and thus create strong correctness guarantees.

Many tools have been developed to enable this by providing logical systems that support such proofs, or by enabling automated proof generation in an automated fashion. One of the earliest tools was *Hoare Logic* (HL) [6], which allowed proving properties of individual executions, so-called trace properties. They include, for example, describing the behavior of a loop or a function in an arbitrary program to verify that the implementation adheres to its intended purpose. More advanced examples based on HL are Viper [11], Dafny [7], or F* [15].

However, HL is limited to properties describing individual executions. *Hyperproperties* [2], properties relating multiple program executions, cannot be verified. These express a variety of useful properties, such as functional and security properties. Thus, new systems were developed to overcome the limitation of trace properties, including Hyper Hoare Logic (HHL) [4].

HHL is a generalization of HL that allows for relating multiple executions of programs in an overapproximative and underapproximative fashion, and further supports arbitrary combinations of these. Overapproximating the set of possible executions can verify properties that should hold for all pairs of executions of the same program. They include properties such as determinism (two executions of a program with the same input will result in the

same output). These are called k-safety- or $\forall^*$-hyperproperties. Alternatively, underapproximating can show the existence of a specific execution—for example, the reachability of an unwanted program state like a bug. Properties of this schema are known as $\exists^*$-properties. HHL was developed to overcome the limitations of existing program logics for hyperproperties such as Relational Hoare Logic [1] and Incorrectness Logic [12], which did not support properties consisting of arbitrary alternations of $\forall^*$- and $\exists^*$-properties [4].

Automation is crucial for logic to be usable in a general setting, which is usually achieved using a deductive program verifier (*verifier* in short). In their recent paper, Dardinier et al. [3] present a new verifier called Hypra, which is based on HHL and extends it. It is the first deductive verifier to support all $\forall^*$-, $\exists^*$-, $\forall^*\exists^*$-, and $\exists^*\forall^*$-properties, and further allows mixing different types of hyperproperties in proofs. Additionally, it is capable of reasoning about runtime errors, allowing us to prove the correctness and incorrectness of a program. It is based on the idea of encoding hyperproperties and the corresponding proof rules in Viper [11], an intermediate verification framework.

In its current state, Hypra has certain limitations. First, it only supports integer-typed variables, significantly restricting the variety of supported programs and properties. For example, properties involving sequences occur in many programs that could be supported by formal verification, but are currently excluded. Second, many error messages are currently imprecise, complicated, and disconnected from the source code. They thus reduce debugging efficiency and negatively affect usability.

The goal of this thesis is to overcome these limitations by extending the set of supported types and designing and implementing better error messages. Our new types include booleans, sequences, sets, and maps with commonly used operations. Additionally, we extend Hypra's suite of benchmarks used to evaluate its capabilities with test cases concerning our new types. Error messages are designed to be precise, informative, and actionable. We propose a system to which the new messages adhere and implement them in Hypra. Finally, we present the development of a Visual Studio Code extension that provides features like syntax highlighting, diagnostics (error detection and visual feedback), auto-completion, and a straightforward installation and execution process.

This thesis is organized as follows. First, we introduce the necessary background about HHL, Hypra, and Viper in Section 2. The implementation and evaluation of our new types are discussed in Chapter 3. Then, Chapter 4 concerns the design and implementation of our improved error messages, and Chapter 5 presents our Visual Studio Code extension. Finally, we conclude and discuss some opportunities for future work in Chapter 6.

Chapter 2

Background

In this chapter, we provide some background required for the subsequent topics. We briefly introduce Hyper Hoare Logic [4], Hypra [3], and Viper [11].

2.1 Hyper Hoare Logic

First, we discuss *Hyper Hoare Logic* (HHL). HHL belongs to the Hoare logic family and can prove arbitrary hyperproperties, including those given in the introduction and combinations. It can prove strictly more properties than other existing program logics. HHL's key idea is to treat assertions as properties of arbitrary sets of reachable states. Assertions can quantify universally over states to show overapproximation and existentially for underapproximation. Combinations allow proving properties that include both. Proofs are performed by establishing hyper-triples of the form $[P]\ C\ [Q]$, where P and Q are pre-/postconditions in the form of hyper-assertions, and C is a program statement. A triple is valid if and only if, for any set of initial states S that satisfies P, the set of all final states that can be reached by executing C in some state from S satisfies Q. Verifying the validity of a hyper-triple proves that the program statement C adheres to its specification described in P and Q [4]. The exact definitions are listed below. We directly cite a definition for hyper-triples that includes errors and originates from Dardinier et al. [3], as it is used in Hypra.

Definition 2.1 (Hyper-triples with errors) *Given a program statement C and two program states σ and σ', we write $\langle C, \sigma\rangle \to \sigma'$ to express that executing C in the initial state σ may lead to the final normal state σ'.*

Executions that lead to runtime errors (because of violated assertions) are denoted as $\langle C, \sigma\rangle_{er} \to \sigma'$, where σ' is the last state reached before the error occurred. We refer to such states as error states, in contrast to normal states.

The set of normal states reachable by executing C in any state from S is denoted as:

$$sem(C, S) \triangleq \{\sigma' \mid \exists \sigma \in S.\ \langle C, \sigma \rangle \rightarrow \sigma'\}$$

while the set of error states reachable by executing C in any state from S is denoted as:

$$err(C, S) \triangleq \{\sigma' \mid \exists \sigma \in S.\ \langle C, \sigma \rangle_{er} \rightarrow \sigma'\}$$

Hyper-assertions are predicates over pairs of sets of states, where the first set corresponds to normal states, and the second set corresponds to error states.

Given two hyper-assertions P and Q, we write:

$$\models [P]\, C\, [Q]$$

to express that the triple $[P]\, C\, [Q]$ is valid, which is defined as follows:

$$\models [P]\, C\, [Q] \quad \textit{iff} \quad \forall S.\ P(S, \emptyset) \Rightarrow Q(sem(C, S), err(C, S))$$

2.2 Hypra

Hypra automates the generation of verification conditions for HHL and thus can be used as a verifier for it. As in HHL, properties are described by assertions that quantify explicitly over states. Verification is attempted by encoding specifications and proof rules into a standard intermediate verification language based on the Viper infrastructure [11]. This is possible by "representing sets of states of the input program explicitly in the states of the intermediate program" [3]. The underlying verifier then performs the verification by checking the validity of the hyper-triple containing the program specification. Hypra's capabilities include reasoning about $\exists^*$- and $\forall^*$-properties and combinations of both. Additionally, Hypra supports reasoning about runtime errors, allowing one to prove the (non-)existence of errors in a program, thus extending the original capabilities of HHL [3].

Hypra verifies properties by establishing hyper-triples [3]. To demonstrate verification using Hypra, let us look at the example program in Listing 2.1. It contains a short program annotated with specifications to show determinism. For this, we construct a hyper-triple of the form

$$[\forall \langle \sigma_1 \rangle, \langle \sigma_2 \rangle.\ \sigma_1(\text{inp}) = \sigma_2(\text{inp})]\ \text{out} := \text{inp} + 10\ [\forall \langle \sigma_1 \rangle, \langle \sigma_2 \rangle.\ \sigma_1(\text{out}) = \sigma_2(\text{out})]$$

using the keywords **requires** to denote preconditions and **ensures** for post-conditions.

Listing 2.1: An example program with the specifications for determinism.

```
method determinism(inp: Int) returns (out: Int)
    requires forall <_s1> ::
        forall <_s2> :: _s1[inp] == _s2[inp]
    ensures forall <_s1> ::
        forall <_s2> :: _s1[out] == _s2[out]
{
    out := inp + 10
}
```

Given a program's pre- and postcondition, Hypra generates a Viper program whose validity implies the validity of the corresponding hyper-triple. This is possible by tracking "the sets of normal states and error states that can be reached by executing C in any state from S [...], and checks whether they satisfy the postcondition Q [...]" [3]. This is done by tracking an upper bound and a lower bound separately. The specifics are omitted here, as they are irrelevant for this thesis. The Viper encoding's main purpose is to construct the set of reachable states. It encodes the pre- and postconditions and then updates the set of reachable states for each atomic statement in C. For this, encodings for all atomic statements are pre-defined and applied as necessary during the generation of the Viper program. For example,

$$\textbf{assume } \forall \sigma_{i+1} \in S_{\forall}^{i+1} \cdot \exists \sigma_i \in S_{\forall}^{i} \cdot \sigma_{i+1} = \sigma_i \, [\text{out} := \sigma_i(\text{inp}) + 10]$$

encodes the assignment from our previous example for updating the lower bound and

$$\textbf{assume } \forall \sigma_i \in S_{\exists}^{i} \cdot \sigma_i \, [\text{out} := \sigma_i(\text{inp}) + 10] \in S_{\exists}^{i+1}$$

for the upper bound, where $S_{\forall}^{i}$, $S_{\forall}^{i+1}$, $S_{\exists}^{i}$, and $S_{\exists}^{i+1}$ are sets of states, and $\sigma[x := v]$ denotes the state σ updated with x set to v.[1] These encodings update a lower bound and an upper bound of reachable states, using the sets of states $S_{\forall}$ and $S_{\exists}$. By reasoning about these sets, we can verify the validity of our hyper-triple, as introduced before. Creating a Viper file in this fashion ensures the equivalence between the validity of the hyper-triple and the validity of the Viper program, thus allowing us to prove specifications written in HHL.

Practically, Hypra consists of an input language and a verifier instance. The input language is designed to cover common use cases that occur in formal verification and includes deterministic and non-deterministic elements. It can be seen in Figure 2.1. Our implementation first performs various checks

[1] The exact definitions and derivations of these formulas are omitted here, as they are not necessary for this thesis. The purpose of the example is to demonstrate the general idea of how encodings are used. Details are provided in Dardinier et al. [3]

C ::= **skip** | **assume** b | **assert** b | C;C (sequential composition)
| x := nonDet() (non-deterministic assignment)
| x := e (assignment)
| **if** (b) C **else** C (conditional)
| **while** (b) C (loop)
| y1, y2, ..., yk := m(x1, x2, ..., xn) (method call)

Figure 2.1: "Input language of Hypra, where C ranges over program commands, x, x_i and y_i range over program variables, e ranges over arithmetic expressions, b ranges over boolean expressions, and m represents a method with n parameters and k return variables" [3].

on the input program, such as symbol, type, and HHL-specific checks. If a program is syntactically and semantically correct according to the specifications of our input language, its corresponding Viper encoding is generated as described above. This program is passed to an internal Viper backend, which attempts to verify it. We analyze the result of the backend, and the corresponding feedback is provided to the user.

2.3 Viper

Viper [11] is an extensive verification infrastructure that was developed for permission-based reasoning. It offers an intermediate language, tool support, and two back-end verifiers based on symbolic execution and verification condition generation. The exact details of Viper are not relevant here, except that it can be used to build new program verifiers on its basis. An example of this is Hypra.

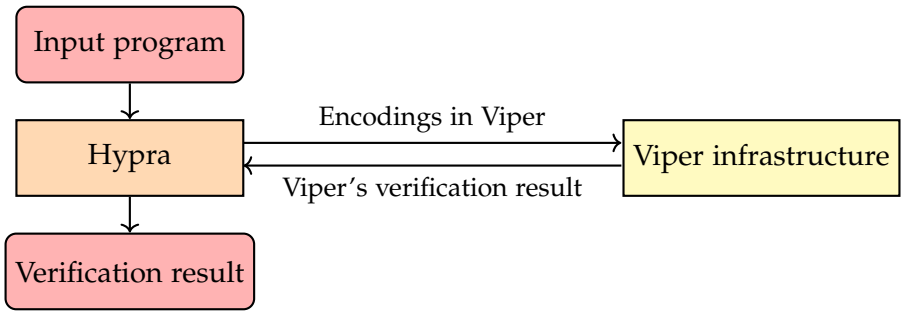

Figure 2.2: Hypra's interaction with the Viper infrastructure. Red nodes indicate input/output, orange ones internal, and yellow ones external components.

When Hypra attempts to verify a program, it generates encodings in Viper's intermediate language. Both Viper backends are then used to verify the generated Viper encoding. We use the result generated by Viper to provide feedback about the verification to the user. Figure 2.2 illustrates this.

Chapter 3

Support of New Types

In this chapter, we introduce four new types to Hypra. Hypra currently only supports the integer type, and the number of supported programs is quite restricted. This excludes many interesting properties that rely on more complex types like sequences and sets, as demonstrated in Listing 3.1, and places Hypra at a disadvantage compared to other verifiers, such as Descartes [13].

To mitigate this, we introduce four new types to Hypra: booleans, sequences, sets, and maps. The latter three are composite types that support generic element types. Additionally, multiple operations are added to provide commonly needed functionality, such as membership tests for sets or element access for maps and sequences.

Our types are introduced in a stepwise fashion, introducing each type with its syntax, semantics, and operations. We start with booleans and cover the types based on the complexity of their implementation. This allows us to introduce required modifications as needed. The types and their operations require encodings for the translation into Viper's input language, which are introduced alongside their types. Furthermore, modifications were made to Hypra's state encoding, and a type translation system based on monomorphization was developed.

The chapter starts by introducing booleans in Section 3.1. Section 3.2 then presents sets, and Section 3.3 maps and sequences. Section 3.4 will briefly touch on the implementation of our work.

Additionally, we evaluate the support for these new types by broadening the suite of benchmarks used to evaluate Hypra. The results allow us to ensure correctness and estimate the impact of our modifications on its performance. Details are provided in Section 3.5.

Listing 3.1: A Hypra program containing sequences to prove that finding the minimal element in a sequence of integers is deterministic.

```
method deterministicMinimum(seq: Seq[Int]) returns
 (out: Int)
    requires forall <_s1>, <_s2> :: _s1[seq] ==
        _s2[seq]
    ensures forall <_s1>, <_s2> :: _s1[out] ==
        _s2[out]
{
    var i: Int
    var min: Int
    min := seq[0]
    i := 1
    while (i < |seq|)
        invariant forall <_s1>, <_s2> ::
            _s1[i] == _s2[i] &&
            _s1[min] == _s2[min] &&
            _s1[seq] == _s2[seq]
    {
        if (seq[i] < min) {
            min := seq[i]
        }
        i := i + 1
    }
    out := min
}
```

3.1 Adding Support for Booleans

We first introduce the boolean type `Bool`, which encodes truth values in Hypra. It can be created by using `var b: Bool` and accepts the two constant truth values `true` and `false`. The supported operations are the logical functions and (`&&`), or (`||`), and not (`!`) with standard semantics. Additionally, implications are supported. Booleans are the only new *primitive* type.

To implement a new type, multiple things need to be modified in our implementation. First, we need to find an equivalent type for the encoding, which gets assigned to values on the Viper level. Every value in our input program has a known type. After the translation, the value still exists in some form and requires a type on its own. For example, let us assume we declare and initialize a variable in the input program by `var b: Bool` and `b := true`, which will be encoded as

$$\textbf{assume } \forall \sigma_i \in S_\exists^i \cdot \sigma_i\,[\mathrm{b} := \mathrm{true}] \in S_\exists^{i+1}.$$

where $S_{\exists}^{i}$, and $S_{\exists}^{i+1}$ are sets of states, and $\sigma[x := v]$ denotes the state σ updated with x set to v. We omit the encoding for the lower bound here and in the rest of this chapter. It does not immediately follow from the mathematical expression, but $\sigma_{i+1}(\mathrm{b})$ carries the value `true`, which must have a type. Naturally, we assign it the type of a Viper boolean. Hypra's boolean presents a simple case, as the axiomatic semantics of Viper's boolean type are fully compatible with ours. This also allows us to translate the previously mentioned operations to their Viper equivalents. For example, the expression `!(true || (false && true))` contains multiple boolean operations. After the translation, the expression will look alike, as all operations were replaced by their corresponding Viper operations, which use the same syntax.

The previously shown encoding also demonstrates how variables and assignments are supported in Hypra. If a value gets assigned to a variable, the value of this variable gets updated in all corresponding states, denoted by $\sigma_i\,[\mathrm{b} := \mathrm{true}]$. This omits some details of the encoding of Hyper Hoare Logic, but this is not further relevant here, as only the states must be modified to support our new type.

Listing 3.2: The original implementation of the *State* domain in Viper.

```
domain State[T]  {
  function get(s: State[T], x: Int): T
  function equal_on_everything_except(s1: State[T],
    s2: State[T], x: Int): Bool

  axiom equal_on_everything_except_def {
    (forall s1: State[T], s2: State[T], x: Int ::
      { equal_on_everything_except(s1, s2, x) }
      equal_on_everything_except(s1, s2, x) ==>
      (forall y: Int ::x != y ==> get(s1, y) ==
        get(s2, y)))
  }
}
```

We now focus on the implementation of this state domain, which is shown in Listing 3.2. The domain provides two functions and one axiom describing its behavior. The function `get(s, x)` provides access to individual elements in the state and allows us to read or set the value of a variable. The function `equal_on_everything_except_def(s1, s2, x)` allows to verify that two states contain identical values for all variables except `x`. Finally, the axiom `equal_on_everything_except_def` encodes a predicate which is used in multiple encodings of Hypra's atomic statements, for example in `havoc` and assignments. The semantics of these elements are not relevant, and we focus on their implementation. As can be seen, the domain is implemented with a generic type parameter `T`, which is used to describe the type of variables

preserved in the state. Our original idea was to create a state for each type that occurs in a program. However, we decided to disregard this approach due to implementation-related issues and instead bundle all types in one state.

Our new implementation of the state domain supports preserving variables of different types in one state. This is possible by replacing the functions and axioms with type-specific versions. If a variable is translated, the type is read during the translation process, and the corresponding function is selected from the domain. The semantics of each method remain unchanged, but each function now supports one specific type instead of a generic parameter. Our new implementation can be seen in Listing 3.3. Additional changes are not required, as the rest of the implementation is compatible with our modified state. This allows us to store both boolean- and integer-typed variables in one state.

Listing 3.3: The new implementation of the *State* domain in Viper with type-specific functions and axioms.

```
domain State  {
  function get_int(s: State, x: Int): Int
  function get_bool(s: State, x: Int): Bool
  function equal_on_everything_except(s1: State,
    s2: State, x: Int): Bool

  axiom equal_on_everything_except_def_int {
    (forall s1: State, s2: State, x: Int ::
      { equal_on_everything_except(s1, s2, x) }
      equal_on_everything_except(s1, s2, x) ==>
      (forall y: Int ::x != y ==> get_int(s1, y) ==
        get_int(s2, y)))
  }

  axiom equal_on_everything_except_def_bool {
    (forall s1: State, s2: State, x: Int ::
      { equal_on_everything_except(s1, s2, x) }
      equal_on_everything_except(s1, s2, x) ==>
      (forall y: Int ::x != y ==> get_bool(s1, y) ==
        get_bool(s2, y)))
  }
}
```

To summarize, we can support booleans by specifying a type to represent values of Hypra's boolean type and modifying the implementation of our states.

3.2 Adding Support for Sets

Our second type is the `Set[T]` type, which describes an unordered set of elements. It belongs to our newly introduced *composite types*. Composite types are types that contain multiple elements of a specific element type, such as a set of integers or a set of sets of integers. As they support generic element types, they can include any type as elements.

Again, let us introduce sets by example using Listing 3.4. Sets can be declared by using `var set: Set[T]`, where `T` is an arbitrary type. They can be created with the syntax `Set[T](`e_1`, `e_2`, ...): Set[T]`, where `T` is an arbitrary type again and e_i are expressions or variables of type `T`. They may also be declared without any arguments to create an empty object. Additionally, the following operations are supported: size retrieval, denoted by the syntax `|set|: Int`, union, intersection, and setminus by `set1 op set2: Set[T]`, and membership check by `e in set: Bool`, where `set`, `set1`, and `set2` have type `Set[T]` with an arbitrary type parameter `T`, `op` is `union`, `intersection` or `setminus` and `e` is an expression or variable of type `T`.

Listing 3.4: A example program demonstrating the usage of sets in Hypra.

```
method testSets(inp: Int) returns ()
requires forall <_s> :: true
ensures forall <_s> :: true
{
    var set: Set[Int]
    set := Set[Int]()
    set := Set[Int](1,2,3,4)
    hyperAssert forall <_s> :: |_s[set]| == 4

    var s1: Set[Int]
    var s2: Set[Int]
    var setOfSets: Set[Set[Int]]
    s1 := Set[Int](1,2)
    s2 := Set[Int](2,3)
    setOfSets := Set[Set[Int]](s1, s2)

    var sun: Set[Int]
    var sin: Set[Int]
    var sdiff: Set[Int]
    sun := s1 union s2
    sin := s1 intersection s2
    sdiff := s1 setminus s2
}
```

As with booleans, we use Viper's set type to encode our sets. Viper's type is again fully compatible with our encoding. The translation is possible due to Viper's set type supporting generic element types. We can thus translate these types recursively, by encoding Hypra's `Set[T]` by Viper's `Set[T']`, where `T'` is the translated `T` type. Thus, Viper's set operations are also compatible.

However, this implementation of sets is incompatible with our previous implementation of states. As described in Section 3.1, we need to need to include specific functions and axioms for each type. Thus, we need to implement them for sets as well. As two set types are equal if and only if their element types are equal, and arbitrary element types are supported, a program can contain an arbitrarily large number of different set types, which cannot be known beforehand. An issue like this could be fixed using a function with generic types. Unfortunately, Viper does not support this.

We mitigate this issue by proposing a *scanning technique* which is closely related to monomorphization. Monomorphization is the process of converting generic functions into specialized versions for each concrete type used [8]. During the translation of a program, we collect all element types that are syntactically used in sets. We then create the functions and axioms for those types during translation. Each function and axiom has a generic version of the form

```
function get_{TID}(s: State, x: {T}): {T}
```

and

```
axiom equal_on_everything_except_def_{TID}
{
    (forall s1: State, s2: State, x: Int ::
      { equal_on_everything_except(s1, s2, x) }
      equal_on_everything_except(s1, s2, x) ==>
      (forall y: Int :: x != y ==> get_{TID}(s1, y) ==
        get_{TID}(s2, y)))
}
```

where `{T}` is replaced by the individual type and `{TID}` by the type's name.

For example, our program from Listing 3.4 contains the set types `Set[Int]` and `Set[Set[Int]]`. The generated encoding of our state for this program can be seen in Listing 3.5. It contains the corresponding functions and axioms for both set types. This allows us to generate the required elements.

In conclusion, sets are implemented by using Viper's set type, translating the types recursively, and using our proposed scanning technique to generate all required elements in our state domain.

Listing 3.5: The encoding of the state domain for the program given in Listing 3.4.

```
domain State  {
  function get_set_int_(s: State, x: Int): Set[Int]
  function get_set_set_int__(s: State, x: Int):
    Set[Set[Int]]
  function get_int(s: State, x: Int): Int
  function equal_on_everything_except(s1: State,
    s2: State, x: Int): Bool

  axiom equal_on_everything_except_def_set_int_
  {
    (forall s1: State, s2: State, x: Int ::
      { equal_on_everything_except(s1, s2, x) }
      equal_on_everything_except(s1, s2, x) ==>
      (forall y: Int :: x != y ==>
        get_set_int_(s1, y) == get_set_int_(s2, y)))
  }

  axiom equal_on_everything_except_def_set_set_int__
  {
    (forall s1: State, s2: State, x: Int ::
      { equal_on_everything_except(s1, s2, x) }
      equal_on_everything_except(s1, s2, x) ==>
      (forall y: Int :: x != y ==>
        get_set_set_int__(s1, y) ==
            get_set_set_int__(s2, y)))
  }

  axiom equal_on_everything_except_def_int
  {
    (forall s1: State, s2: State, x: Int ::
      { equal_on_everything_except(s1, s2, x) }
      equal_on_everything_except(s1, s2, x) ==>
      (forall y: Int :: x != y ==> get_int(s1, y) ==
        get_int(s2, y)))
  }
}
```

3.3 Adding Support for Maps and Sequences

Finally, we add support for maps, collections of key-value pairs where each key is unique, and sequences, ordered lists of elements.

Maps with the type `Map[S, T]` can be declared by using `var map: Map[S, T]`, where `S` and `T` are arbitrary types. Creation is possible by `Map[S, T](`k_1 `:=` v_1`,` k_2 `:=` v_2`, ...): Map[S, T]`, where `S` and `T` are defined are arbitrary types and k_i are expressions or variables of type `S` and p_i of type `T`. Maps support element access by `map[k]: T`, element updates by `map[k := v]: Map[S, T]` and membership checks by `k in map: Bool`, where `map` is of type `Map[S, T]`, `k` is a key of type `S` and `v` is a value of type `T`.

Sequences have the type `Seq[T]` and can be declared by using `var seq: Seq[T]`, where `S` is an arbitrary type. They can be created by `Seq[T](`e_1`,` e_2`, ...): Seq[T]`, where `T` is defined as an arbitrary type and e_i are expressions or variables of type `T`. Supported operations include element access (accessing the element at a specific index) and concatenation (appending a sequence to another one), denoted by `seq1[ind]: T` and `seq1 ++ seq1: Seq[T]`, where `seq1` and `seq2` are arbitrary sequences of type `Seq[T]`and `ind` is an expression of type `Int`.

As sequences and maps support generic type parameters for their elements, we apply the scanning technique defined in the previous section. This ensures the compatibility between these types and our state domain.

However, unlike the previous types, sequences and maps cannot be directly translated to their Viper equivalents. This is due to an incompatibility in the axiomatizations underlying these types in Hypra and Viper. As mentioned in the introduction, Hypra supports reasoning about errors. We refer to these as runtime errors to distinguish them from errors related to verification failures. However, Viper has no equivalent to this. So far, this has not affected the encoding of our new types, as none of the previously introduced operations could result in an error. This has changed by sequences and maps supporting the element access operation. If an index is out of bounds or a key does not exist in a map, the expression is not well-defined. Hypra treats this as a runtime error, while Viper fails to verify the program. An example of this can be seen in Listing 3.6.

$$\textbf{assume } \forall \sigma_i \in S_\exists^i \cdot \sigma_i\,[\text{seq} := \text{Seq[Int]}(1,2,3)] \in S_\exists^{i+1}$$
$$\textbf{assume } \forall \sigma_{i+1} \in S_\exists^{i+1} \cdot \sigma_{i+1}\,[\text{out} := \sigma_{i+1}\,[\text{seq}]\,[\sigma_{i+1}\,[\text{ind}]]] \in S_\exists^{i+2}$$

Figure 3.1: An excerpt from the encoding of the program in Listing 3.6 showing the creation of the sequence and the element access operation. All mentioned types belong to Viper's type system. i is a natural number used to enumerate created states.

Listing 3.6: A Hypra program to prove that the element access operation on sequences of integers is deterministic. It contains a sequence with an element access operation that may result in an out-of-bounds runtime error. This program can not be verified using the naive encoding based on Viper's sequence type.

```
method outOfBounds(ind: Int) returns (out: Int)
    requires forall <_s1>, <_s2> :: _s1[ind] ==
        _s2[ind]
    ensures forall <_s1>, <_s2> :: _s1[out] ==
        _s2[out]
{
    var seq: Seq[Int]
    seq := Seq[Int](1,2,3)
    out := seq[ind]
}
```

This program should verify according to Hypra's specification. However, if we use a naive encoding based on Viper's sequence type, this program fails to verify, because the well-definedness is not guaranteed. The naive encoding can be seen in Listing 3.1. Thus, we need to provide custom axioms for sequences and maps.

We implement these types by creating two new Viper domains, called *HHLSeq* for sequences and *HHLMap* for maps.Their encodings resemble these of their original Viper parts closely, with modifications made to prevent the previously mentioned issues. This is achieved by excluding axioms that check if an index or key is valid and some small changes ensuring compatibility with Viper's remaining types. The domains also include all relevant functions required to support their operations. They can be found in our repository[1] and are inspired by Leino [7]. When a sequence or map type is detected during compilation, the corresponding domains will be added to the encoding. During translation, these domains are used as the types representing sequences and maps.

Finally, our encoding needs to ensure that an invalid access in a sequence or map results in an error state to support Hypra's reasoning about errors. This is implemented by detecting statements that include any element access

[1] https://github.com/anqili426/hhl_frontend

Hypra type	Viper counterpart
Int	Int†
Bool	Bool†
Seq[T]	HHLSeq[T]
Set[T]	Set[T]†
Maps[S, T]	HHLMap[S, T]

Figure 3.2: Newly implemented Hypra types and their corresponding types on the Viper level. † indicates a direct mapping between a Hypra and a Viper type.

$$\textbf{assume } \forall \sigma_i \in S_{\exists}^{i} \cdot \sigma_i\,[\text{seq} := \text{HHLSeq[Int]}(1,2,3)] \in S_{\exists}^{i+1}$$
$$\textbf{assume } \forall \sigma_{i+1} \in S_{\exists}^{i+1}.0 \leq \sigma_{i+1}(\text{ind}) \leq \text{HHLSeq\_length}(\sigma_{i+1}(\text{seq})) \Rightarrow \sigma_{i+1} \in S_{\exists}^{i+2}$$
$$\textbf{assume } \forall \sigma_{i+1} \in S_{\exists}^{i+1}.\neg(0 \leq \sigma_{i+1}(\text{ind}) \leq \text{HHLSeq\_length}(\sigma_{i+1}(\text{seq}))) \Rightarrow \sigma_{i+1} \in S_{\exists}^{\perp,\, i+2}$$
$$\textbf{assume } \forall \sigma_{i+2} \in S_{\exists}^{i+2} \cdot \sigma_{i+2}\,[\text{out} := \sigma_{i+2}\,[\text{seq}]\,[\sigma_{i+2}\,[\text{ind}]]] \in S_{\exists}^{i+3}$$

Figure 3.3: An excerpt from the improved encoding of the program in Listing 3.6 showing the creation of the sequence and the element access operation. `HHLSeq` denotes the custom sequence type, `Int` Viper's int type. i is a natural number used to enumerate created states. $S_{\exists}^{i}$ are sets of states and $S_{\exists}^{\perp,\, i}$ are sets error states. The second statement ensures that the index is in bounds for all normal states, and the third statement places all other cases in the error states.

on these types during type-checking and marking them. When a marked statement is translated, its encoding is slightly modified to assert the in-bounds or membership check before the element access is performed. A failed assertion will thus result in an error state, as specified by Hypra's encodings [3]. Our updated encoding is shown in Figure 3.3. This is implemented in the same fashion for maps, but the encoding is omitted here.

To summarize, adding new types to Hypra consists of three elements. First, a mapping between types in the input program and the types representing them in the encoding. Those types have been introduced individually during the previous sections, but a summary can be found in Figure 3.2. Second, we modify the implementation of our states to preserve values of arbitrary types. This is supported by our scanning method, which generates support functions dynamically for all syntactically used types. Finally, not all types can be mapped to a Viper type directly. We thus introduce two new domains, HHLSeq and HHLMap, in Viper, which closely resemble the encoding of Viper's types. Nonetheless, their encoding and their operations have been modified to support Hypra's reasoning about runtime errors.

3.4 Implementation Details

The implementation of the new types is rather straightforward, as we solely extended the existing architecture of our application. Therefore, most details are omitted here, and interested readers should be able to understand our changes based on the project's documentation.

When designing our new types, we tried to mimic the design used by Viper as closely as possible. The types were selected based on common use cases for Viper and their syntax, semantics, and further attributes, such as precedence, follow Viper's standards. Specifically, we aimed to make working with these types in Hypra as natural as possible for users familiar with our systems. As of now, not all operations that are supported by Viper for these types are implemented, but the implementation can easily be modified in the future.

Introducing new types to Hypra also requires modification to our error-handling procedure, as various new type errors are now possible. These are treated as internal errors and their handling follows the guidelines described in Section 4.1. This requires a context consisting of their category, cause, and position. For expressions, all required information is already provided (see Section 4.4), but statements and consequently variable assignments lack this information. Thus, all statements that can include type errors were modified to preserve the same information as expressions. This allows us to build error messages independent of the error's cause.

3.5 Testing and Evaluation

When complex modifications such as the implementation of new types are performed, thorough testing is required to ensure correctness and compatibility with the system. Testing of our new types is performed in two separate steps: First, custom test cases were used that test the behavior of our new types specifically. This allows us to check the correctness of our encoding and catch bugs that were introduced during the implementation phase. Second, we extended Hypra's benchmark suite by implementing cases that were not supported by the current type system. Hence, we can assess the impact of our modifications on Hypra's performance and demonstrate its capability to reason about test cases involving our new types.

3.5.1 Type-Specific Tests

Type-specific tests are focused on checking the correctness of our implementation. Thus, they are not concerned with aspects relating to the evaluation of Hypra itself. They are composed of four different test files, one for each newly implemented type, that cover features associated with each type. They allow us to assess whether specific operations, the types themselves, and

Benchmark	Type of property	File (LoC)	Verification time mean (s)	Annotations (LoC)
ArrayIntFalse (P1)	$\forall^*$	30	3.1	7
ArrayInt (P1)	$\forall^*$	41	3.2	7
ArrayInt$^-$ (P2)	$\forall^*$	42	12.2	20
ArrayInt$^-$ (P3)	$\forall^*$	42	12.7	9
ChromosomeFalse (P1)	$\forall^*$	56	53.5	0
Chromosome (P1)	$\forall^*$	30	2.7	8
NzbFile (P1)	$\forall^*$	66	3.4	14
Word (P1)	$\forall^*$	44	5.7	8
NzbFileFalse† (P1)	$\exists^*$	72	1.6	0

Figure 3.4: List of our newly implemented benchmarks, including the results for their execution. † indicates benchmarks with unsatisfied properties. These were verified by strengthening the preconditions and negating the postconditions. $^-$ declares that a benchmark is simplified and the specific property is denoted in the brackets. LoC is short for lines of code.

other aspects are translated correctly into their Viper encoding and test common features. All files can be found in the repository and are listed in the appendix. An excerpt of the test file for sequences can be seen in Listing 3.4.

3.5.2 Benchmarks and Evaluation

The benchmarks evaluate Hypra's performance regarding its previous implementation and other state-of-the-art verifiers, such as Descartes [13]. As mentioned, the evaluation should provide insights into the performance impact of our modifications and demonstrate Hypra's capabilities.

Implementation	Mean (s)	Median (s)
Old Implementation	2.8	1.4
New Implementation	4.8	1.4

Figure 3.5: Comparison of Mean and Median of the execution time for the old and new implementations. The analysis was only performed on benchmarks supported by both implementations.

We broadened Hypra's benchmark suite by implementing a subset of previously excluded benchmarks due to unsupported types. These test cases were restricted to the publicly available benchmarks of Descartes' Stackoverflow suite [13]. As described in Dardinier et al. [3], we translated the programs into Hypra's syntax and obtained semantically equivalent program specifications. We also annotated loop invariants and loop variants where necessary. In total, we translated eight benchmarks including $\forall^*$-hyperproperties, and disproved one $\forall^*$-hyperproperty by forming its negation using $\exists^*$-hyperproperties. Two of the newly implemented benchmarks could only be proven in a simplified version (indicated by $-$).[2] An evaluation of all newly implemented benchmarks can be seen in Figure 3.4. In total, 92 benchmarks were evaluated, including 83 original and nine new ones. Our results, including all benchmarks, can be seen in Figure 3.6. For some unknown reason, one previously

[2] A simplified version proves specifications weaker than the originally proposed ones.

Type of hyperproperty	Source	Files		Verification time		Annotations
		no.	Mean (LoC)	Mean (s)	Median (s)	Mean (LoC)
$\forall^*$	Descartes	23	86	6.0	2.5	3.2
	PCSat	3	23	1.2	1.2	2.7
	Overall	**26**	**78**	**5.5**	**2.3**	**3.1**
$\exists^*$	Descartes†	9	69	9.5	2.7	0.0
	ORHLE	5	14	2.6	1.5	5.8
	Overall	**14**	**49**	**7.0**	**2.6**	**2.0**
$\forall^*\exists^*$	ORHLE	28	20	8.2	1.8	1.2
	HyPa	8	14	1.4	1.2	2.1
	PCSat	1	22	1.8	1.8	2.0
	Overall	**37**	**19**	**6.5**	**1.7**	**1.4**
$\exists^*\forall^*$	ORHLE†	15	25	2.6	1.3	1.7

Figure 3.6: "Results of our evaluation. Benchmarks marked with † are obtained by strengthening the preconditions and negating the postconditions of the original benchmarks that fail to prove $\forall^*$- or $\forall^*\exists^*$-hyperproperties. We count use statements, loop variants and loop invariants as annotations." [3]

evaluated benchmark could not be verified. Further investigation is needed, as it should not have been affected by our changes. We thus exclude it from this evaluation.

The evaluation was performed on a MacBook Pro with the following specifications: MacOS Sequoia 15.3.1, 2.6 GHz 6-Core Intel Core i7, 16 GB RAM. Each benchmark was run with 3 repetitions.

Our results show that Hypra can indeed verify hyperproperties that reason about programs containing our newly implemented types. Figure 3.4 presents our results for the individual, newly introduced benchmarks. As can be seen, most benchmarks can be proven in a reasonable timeframe and with an acceptable number of proof annotations. However, proving properties that include composite types may require noticeably more runtime and proof annotations than other benchmarks, as demonstrated by *ArrayInt (P1)* and *ChromosomeFalse (P1)*.

Our implementation is also noticeably slower than the previous one, as can be seen in Figure 3.5. It was created by evaluating all previously implemented benchmarks using both implementations and then calculating the mean runtime of all benchmarks. No programs using the newly implemented types were used for this. As the higher mean and equal median indicate, our implementation increased the runtime of certain benchmarks significantly. However, the effect is still acceptable as most benchmarks verify in an acceptable timeframe.

For completeness, we also present the updated evaluation from Darinier et al. [3] in Figure 3.6. It now includes all implemented benchmarks (except the excluded one) and allows a comparison with the original analysis.

We can conclude that Hypra is indeed capable of supporting our new types while requiring an acceptable runtime and number of proof annotations. Nonetheless, some negative effects occurred, which may require further investigation.

Chapter 4

Improvement of Error Messages

The objective of this chapter is to improve error messages generated by Hypra during the verification of an input program. This contributes to our overall goal of improving various aspects of Hypra to enhance its usability. Error messages, which are part of a system's error handling, play a fundamental role in this regard. They are used to provide information to the user about an issue and are therefore the main source of information. Thus, an unsatisfactory or poorly developed error message might significantly inhibit debugging. This is amplified by the fact that Hypra works with Hyper Hoare Logic, and related errors are often the result of advanced logical formulas. Good error handling is, therefore, essential for a smooth user experience.

Hypra's current error messages face certain limitations. They often lack clarity and obscure their underlying cause and context, which are necessary to understand the environment in which the error occurred. This includes aspects such as error category, the position in the code, and information about the encoding used to translate the provided code. The last one is particularly important in the context of while loops, as multiple encodings with different effects are possible. Thus, the messages often lead to a complicated debugging process and can, in some cases, only be understood by an experienced user familiar with the inner workings of our software.

Listing 4.1: An incorrect program containing a while loop with an invariant. Attempting verification on this programs results in an error as the invariant cannot be satisfied.

```
method loopInvariantEntryErr() returns (out: Int)
    requires forall <_s> :: true
    ensures forall <_s> :: _s[out] == 2
{
    var i: Int
    i := 0

    while (i < 1)
        invariant forall <_s> :: false
    {
        i := i + 1
    }

    out := 2
}
```

Listing 4.2: The error message which is created by the previous Hypra instance when the program in Listing 4.1 is executed.

```
Verification failed
Assert might fail. Assertion (forall _s: State[Int]
:: { (in_set_forall(_s, S): Bool) }
!(in_set_forall(_s, S): Bool) || false) might not
hold. (<no position>)
```

Listing 4.3: The new error message which results from executing the program in Listing 4.1.

```
[2025-04-03T14:54:42] [ERROR] Verification Error in
/Users/paulwinkler/Desktop/hhl_frontend/src/test/
temp.hhl:188: The loop invariant forall <_s: state>
:: ((<_s>) ==> (false)) might not hold at entry
point (syncRule)
```

Listing 4.1 demonstrates this by showing an example program. The program contains a while loop with an invariant that cannot be satisfied. Listing 4.2 shows the resulting error message. As can be seen, the error message appears to be disconnected from our original program. It lacks information about its cause, its position, and any possible debugging options. A user needs to know that the verification of while loops includes asserting invariants and that expressions are translated as presented. Otherwise, they would not be able to relate the message to the loop's invariant. Hence, this message is unsatisfactory by common development standards.

We resolve this issue by introducing a standardized system and modifying the verification process to support it. The former defines standards that

enable a user to understand and resolve the underlying error. It includes all aspects that were mentioned earlier. To generate messages adhering to this system, information about the underlying error and the source code is necessary. This might not always be the case, as Hypra interacts with external verification software. We thus modify the verification process to preserve all required information. Both aspects then allow us to generate satisfactory error messages. Finally, the improved error messages are implemented by updating Hypra's error handling to generate suitable messages.

Our results can be seen in Listing 4.3, which presents the new message for our previous example. It now contains all the required information and should allow for improved debugging.

We propose our standardization in Section 4.1 and also outline the general mitigation strategy. Section 4.2 discusses the different origins of errors in Hypra, and Section 4.3 covers the challenges that lead to the mentioned modifications. Our implementation is presented in Section 4.4, and Section 4.5 provides some examples.

4.1 Design of Improved Error Messages

Before we can improve the error messages, it is necessary to define the requirements a good message must fulfill. We propose the following criteria: Error messages should be (1) precise and informative, (2) actionable, and (3) consistent in wording and formatting.

(1) ensures that a user learns about the cause of an error. This should be done in a minimalistic way, while providing as much information as necessary to understand the situation.

(2) guarantees that error messages provide enough context to instruct the user on where and how they should debug the error. Location annotations, categories, and hints/explanations belong to this category.

Finally, we want to ensure that all messages adhere to a shared format, allowing users to become accustomed to our error-handling system and creating a professional appearance. This is achieved by (3).

Listing 4.4: An error message adhering to our design guidelines.

```
[2025-04-19T18:37:40] [ERROR] Verification Error in
/Users/paulwinkler/Desktop/hhl_frontend/src/test
/errors/loopInvariantErr_false.hhl:646: The loop
invariant forall <_s: state> :: ((<_s>) ==> ((_s[I])
== (0))) might not hold (syncRule)
```

With these guidelines, we can now design our actual error messages. Listing 4.4 shows an example. We can see that the message is written precisely and

informatively, ensuring (1). It contains a category (*Verification Error*), the error's cause (*The loop invariant [...] might not hold*), and information about the encoding (*syncRule*). Additionally, it mentions the faulty code snippet and points to its position in the source code, thus offering debug information. Finally, we adhere to (3) by creating all error messages in this way.

Defining the design guidelines forms our first essential step toward better error messages.

4.2 Errors in Hypra

We now focus on the possible origins of errors in Hypra. All possible error sources must be modified so that error messages are generated in adherence to our design guidelines. This poses a challenge, as the implementation varies significantly depending on an error's origin. We thus categorize the errors and discuss their specific challenges.

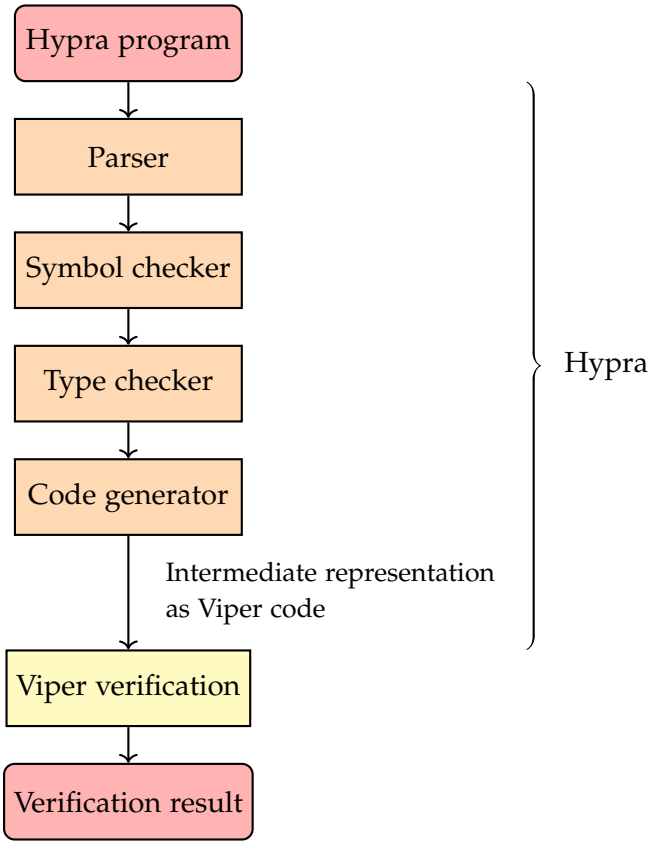

Figure 4.1: Hypra's verification process. Red nodes describe program input and output, orange nodes internal, and yellow nodes external program modules.

To create our categories, we need to understand the different kinds of errors that can occur during a Hypra execution. Let us examine Hypra's verification procedure, shown in Figure 4.1. It outlines the individual steps completed during execution. We can separate these into internal components, which are developed by us and are part of Hypra itself, and external steps, which involve external software used by Hypra. We do not have access to the internal workings of the latter. It can be seen that the internal components essentially act as a source-to-source compiler, converting programs from Hypra's input language into Viper's input language. The converted code can then be verified by relying on a Viper backend. Note that verification is thus handled by external software, not Hypra itself. Errors can occur at any point during the execution, but they differ fundamentally based on their origin. Error handling in the internal components was developed by us and can be easily modified. However, for errors occurring in the external part, we fully rely on Viper's error handling.

4.2.1 Internal Errors

Our first category is internal errors. As mentioned, errors in this category arise in the internal steps and are related to the translation between the input languages. Examples include symbol errors (invalid usage of symbols in the source code), type errors (use of incompatible types in the source code), and verification errors (errors originating from incorrectly used verification specifications). An example containing a type error can be seen in Figure 4.5. Its error message is shown in Figure 4.6, and we note that this message lacks crucial information to identify the faulty precondition. The key point here is that these errors arise from our codebase. Thus, we can improve them by updating the implementation of each error handler. The errors arising in different steps of the translation process are unique to that step. For example, an error arising during type checking is always a type error. We know the necessary information about the cause and can emit good error messages directly. Thus, this is solely an implementation challenge.

Listing 4.5: A program containing an invalid precondition. Preconditions must be assertions containing Hyper Hoare Logic.

```
method illegalPrecondition(inp: Int) returns (out:
    Int)
     requires true
     ensures forall <_s> :: true
{
     out := inp
}
```

Listing 4.6: The error message which is created by the previous Hypra version when the program in Listing 4.5 is executed.

```
At least one precondition of method
 illegalSpecification is not a hyper assertion
```

4.2.2 External Errors

This is not the case for external errors. These occur when the external verification process, conducted by the Viper module, fails. This can happen, for instance, when the provided specifications are not strong enough to prove a specific hoare-triple [4]. When Viper encounters such a verification failure, it returns an error message describing the issue. This behavior is beyond our control, as Viper's internal workings are closed off. We cannot implement different error handlers as we could before, since we only learn about the failure from the returned error message. Therefore, we must generate a custom error message based on Viper's emitted message. So far, we have simply forwarded this message to the end user. Listing 4.2 shows an example of such an error message. As discussed earlier, these messages are often unsatisfactory and appear disconnected from the input program. While we could generate a new error message based on Viper's message, this is not straightforward. It would require a program capable of analyzing these messages and linking them to the provided intermediate representation.

Viper's error messages are limited because they refer to the intermediate representation (IR) rather than Hypra's input, which fails to reflect the original structure. The translation process applies several transformations to the original code. Contextual changes introduced during encoding often increase complexity, as the encoding captures additional aspects related to verification. These factors lead to highly complicated and disconnected error messages.

We now focus on our mitigation strategy, which involves providing annotations in the intermediate representation. If we can annotate the code with additional information that is used in Viper's emitted messages, implementing an error-handling system on top of Viper becomes feasible. However, providing information for every translated element of the original input would significantly increase the complexity of Viper's input, decrease verification efficiency, and result in a highly complex error-handling system, as every possible case would need to be addressed.

We now aim to limit the causes of such errors to a well-defined set of cases. Since Hypra acts as a source-to-source compiler, the IR is the result of a deterministic translation process. Therefore, at a fundamental level, every IR is composed of a finite set of basic building blocks known to us. Additionally, only syntactically correct Hypra programs are translated into an IR. These

constraints restrict the IR to a set of Viper programs that can only fail due to verification errors. This is a crucial insight, as only a limited subset of Viper statements can lead to such failures. Consequently, we can focus on modifying only the relevant code blocks to implement error-handling mechanisms. Further analysis reveals that these errors always stem from boolean expressions that cannot be proven. This significantly reduces the complexity of improving our error-handling system.

We now turn to improving the corresponding error messages. Since Viper generates these messages automatically based on the IR, we must ensure that the IR contains the necessary information to support clearer messages. Fortunately, Viper supports annotations on statements, which are then included in the generated error messages. We leverage this feature by appending annotations to all statements and expressions that might fail during the translation process. If Viper encounters an error, it will return our annotation instead of its default error message, resulting in more informative and satisfactory messages.

Let us apply this principle to an example. Listing 4.7 contains a program, and Listing 4.8 shows a part of its Viper encoding. Based on the reasoning above, we know that Viper is not supposed to fail unexpectedly, as the IR is syntactically correct, and only certain statements may cause errors. In this case, only the postcondition could trigger an error during Viper's verification. Therefore, we annotate the postcondition with the necessary information to generate a message. If Viper fails due to the postcondition, the annotation's information will be returned. The returned data can be analyzed by us to generate the error message.

Listing 4.7: A simple program containing invalid specifications in their postconditions.

```
method postconditionError() returns (out: Int)
requires exists <_s> :: true
ensures exists <_s> :: _s[out] == 0
{
    out := 1
}
```

Listing 4.8: An excerpt from the Viper encoding resulting from the program in Listing 4.7. It shows postcondition's encoding. *@msg* is used to provide annotations. For better readability, the JSON data containing the possible error information was replaced by *[provided error data]*.

```
ensures @msg([provided error data])
((exists _s: State :: in_set_exists(_s, S) &&
get_int(_s, out) == 0))
```

4.3 Challenges

The previous strategy provides a solid foundation for our mitigation approach but has certain limitations. These limitations arise from the process of encoding Hypra's input language into Viper's input language. The two languages verify different logical systems Hyper Hoare Logic (HHL) [4] and permission-based reasoning [11]—and Hypra is built on the idea of verifying HHL by translating it into Viper's system and using Viper as the verification backend. This encoding process poses challenges, as some elements of Hypra's input language require complex translations into Viper. These translated elements introduce statements that may fail but are not part of the original input program, as they are either significantly modified or newly introduced during translation. If one of these statements fails, Viper will emit an error. However, since these errors stem from the translated representation, we cannot directly forward the error information to the user, who may be unaware of these modifications. Therefore, we need to develop specific approaches to handle these cases effectively.

4.3.1 Transformed Expressions

The first case concerns transformations applied to expressions. These may occur when an expression is transformed into a semantically equivalent form, multiple expressions are joined to form a conjunction, or are separated into smaller formulas. The reasons for this vary. The mitigation of this issue is straightforward, as the expressions have a direct origin in the source code. We thus always refer to the original expression if a transformed expression of this kind results in an error.

4.3.2 Modular Verification

Not all statements in Hypra are verified as a single continuous block of code. Instead, the encoding includes modular functions that verify specific aspects independently of the original function. A prime example is a while loop, as seen in Listing 4.9. When a while loop is translated, the encoding performs specific checks to verify the statement and assumes the loop's postconditions in the main method. [1] The loop's invariants, among other aspects, are verified in a separate method dedicated solely to the loop [3].

[1] This is not an exhaustive description.

Listing 4.9: A while loop extracted from a Hypra program. It has invariant of the form $\exists\forall.P$ and P is an arbitrary expression. It also specifies the existsRule, which is selected during its translation.

```
while existsRule (i < n)
invariant exists <_s1> :: forall <_s2> :: P
{
    [...]
}
```

When an error occurs during the modular verification, Viper often describes code that does not exist in the source code. Additionally, the errors raised may have causes that differ from what Viper indicates. For example, Viper might report an unsatisfied postcondition, which is actually related to an invariant, not the input's postcondition. However, every expression causing an error has a corresponding expression in the original encoding. In this case, the invariant would be the origin of the postcondition error. By annotating these expressions with information about their origins, we can mitigate this issue.

4.3.3 Translation of Loop Rules

When while loops are translated, two fundamental problems arise. First, they might include premises specific to the loop rule, and second, multiple loop rules might be applied recursively.

Users can specify the rule applied to a while loop, but often the decision is made automatically during the translation. Without a specified rule, the user does not know which rule was applied. This is a problem, as the rules have different premises that are translated in a modular manner. We know from the previous Subsection that errors from modular methods are mapped back to their origin. This was sufficient before, but additional information about the applied loop rule is required here. Otherwise, users are not able to deduce loop-rule-specific reasons for a failed verification. This is even more important, as the loop rule may apply significant transformations to the expression. Referring to the original expression in this case is a simplification, which is only acceptable when the user can reconstruct the exact situation leading to the error. This is possible by referring to the applied loop rule and Hypra's encodings. Thus, errors thrown by while loops are annotated with the used loop rule.

$$\frac{\forall v.\, \models [\exists\langle\sigma\rangle.\, P_\sigma \wedge b(\sigma) \wedge v = e(\sigma)]\ \textbf{if}\ (b)\ \{C\}\ [\exists\langle\sigma\rangle.\, P_\sigma \wedge e(\sigma) \prec v] \quad \forall\sigma.\, \models [P_\sigma]\ \textbf{while}\ (b)\ \{C\}\ [P_\sigma] \quad \prec \text{wf}}{\models [\exists\langle\sigma\rangle.\, P_\sigma]\ \textbf{while}\ (b)\ \{C\}\ [(\exists\langle\sigma\rangle.\, P_\sigma) \wedge \square(\neg b)]}\ (\mathit{While}{-}\exists)$$

Figure 4.2: Hypra's *existsRule* taken from Dardinier et al. [3].

Listing 4.10: An example of an error emitted when a while loop cannot be verified with the existsRule. The loop invariant adheres to the example in Listing 4.9. The first message concerns the unmodified invariant, the second message describes the modified one.

```
[2025-04-20T10:47:14] [ERROR] Verification Error in
/Users/paulwinkler/Desktop/hhl_frontend/src/test/
errors/existsForallRuleErr_false.hhl:984: The
loop invariant exists <_s1: state> :: ((<_s1>) &&
(forall <_s2: state> :: ((<_s2>) ==>
((_s2[i]) >= (0))))) might not hold at entry point
(existsRule)

[2025-04-20T10:47:15] [ERROR] Verification Error in
/Users/paulwinkler/Desktop/hhl_frontend/src/test/
errors/existsForallRuleErr_false.hhl:991: The
loop invariant exists <_s1: state> :: ((<_s1>) &&
(forall <_s2: state> :: ((<_s2>) ==>
((_s2[i]) >= (0))))) might not hold (1 existential
quantifiers removed) (forAllExistsRule)
```

The latter problem arises when Hypra's *existsRule* is used. The rule is shown in Figure 4.2, and for simplicity, we will discuss it alongside the example seen in Listing 4.9. When this rule is applied, two premises are verified modularly, with the first one being covered by our previous case. [2]

For the second premise, the provided invariant is stripped of its top-level existential quantifier, and the loop is then verified under this modified invariant. This verification is also done modularly. In our example, the method attempts to verify $\forall\sigma.\ |= [\forall s_2\ P_\sigma]$ **while** (b) C $[\forall s_2\ P_\sigma]$. Notice that we are now verifying a while loop with a different invariant than before. Unlike the other rules, *existsRule* is not strong enough to directly verify the loop. Thus, the new and transformed invariant must be proven separately using an additional loop rule. This leads to the loop being treated as a separate entity, and all previously mentioned issues arise again. As a result, it is no longer sufficient to merely refer to the original expression, since the semantics have fundamentally changed.

We mitigate this by keeping track of the transformations applied to the invariant in the new method. Precisely, we track how many quantifiers were

[2] This excludes the premise regarding well-foundedness, which is not relevant here.

removed during the verification process and again annotate the applied loop rule. Should errors exist in applications of multiple recursively applied loop rules, we emit multiple errors. Thus, a user should be able to understand which rule couldn't be verified and in which transformation state the error occurred. An example can be seen in Listing 4.10.

4.4 Implementation

This section covers the implementation of the error messages for internal and external messages.

4.4.1 Error Messages

As mentioned, we want to achieve standardized error messages that adhere to our proposed design guidelines. To do this, we created the three systems shown in Figure 4.3. The exact implementation differs from the structure shown, as we also implemented a standardized logger system. Error handling is a subpart of this new system, but it is not relevant here.

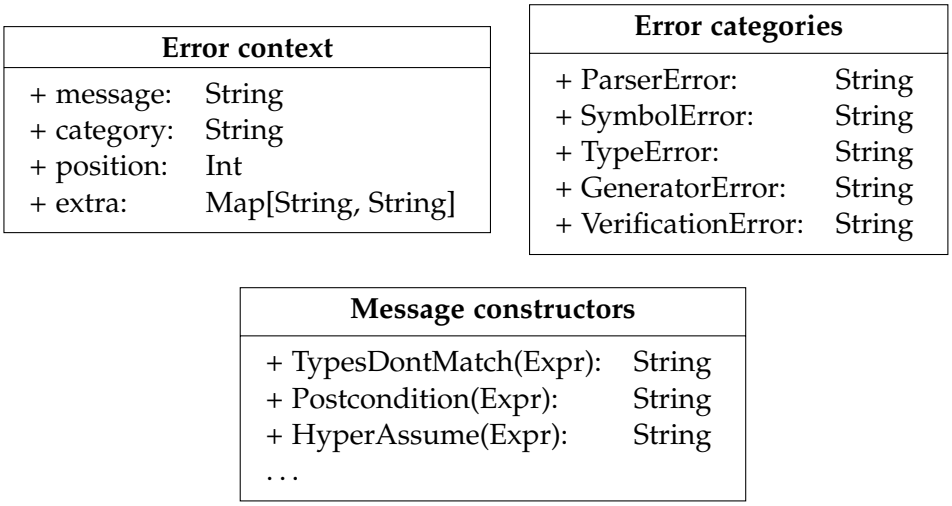

Figure 4.3: Hypra's error context

The error context describes all the required information to generate effective messages, as explained in the previous chapters. Every possible error in Hypra will result in the creation of an object of this kind, which can then be further processed. For example, it could be logged to the console or sent via an API to another program. Since the `extra` attribute might not be straightforward, let us elaborate on it further. It serves as an optional container that can be used to store any information required only by certain

errors. This mostly includes information about while statements and applied loop rules, as discussed in Section 4.3.

Error categories contain constants that are assigned to errors to describe their categories.

Finally, message constructors are functions used to create error messages for specific use cases.

4.4.2 Internal Error Messages

The implementation of better messages for internal errors is straightforward. Due to their nature, we can access the information needed to construct meaningful messages and only need to implement supporting methods and data structures in our code.

4.4.3 External Error Messages

External error messages are created using the same data structures but include the additional step of being converted into a Viper annotation. The error context is generated during the creation of the intermediate representation, passed into Viper, and may be re-emitted by Viper in the case of an error. Since Viper now returns the same data structure describing the error as internal errors, generating the message is straightforward.

To enable the functionalities described in Section 4.3, we need to dynamically track information regarding the mentioned cases. As noted at the beginning of this chapter, every external error is connected to one expression. Therefore, expressions can carry the context, which is updated when one of these cases applies. It thus keeps track of all applied changes, and to access the context of an error, we only need to access its expression.

Our handling of external error messages is based on passing the previously constructed context into our intermediate representation of Viper. This is possible by using an `AnnotationInfo` decoration in Viper. This object allows us to modify the behavior of Viper: Should an *annotated* statement cause an error, the provided data in the `AnnotationInfo` will be emitted instead of an automatically generated message.

4.5 Testing

We also tested the generation of these error messages as part of this chapter. This seems trivial for internal errors, as they follow clearly defined patterns based on the cases they are meant to handle.

For external errors, on the other hand, we had to identify the basic building blocks causing errors and then annotate them. Describing the origin of an error can be challenging, as argued in Section 4.3.

Therefore, we created a test case for every possible error that can occur based on our translation of the IR. Then, we optimized the generated error messages until they reached a satisfactory level.

All test cases can be found in the repository. [3] One example is provided in Listing 4.11 to demonstrate our approach. The corresponding error is shown in Listing 4.10.

Listing 4.11: A incorrect Hypra program containing a while loop which cannot be verified. Based on the invariant, the verification uses the *existsRule* and the *forAllExistsRule* in a chained manner.

```
method existsForallRuleErrla(n: Int) returns ()
requires exists <_s> :: true
ensures exists <_s> :: true
{
    var i: Int
    var r: Int

    havoc i {hint1}
    use hint1(-1)
    use hint1(1)

    while (i >= 0 && i < n)
    invariant exists <_s1> :: forall <_s2> :: _s2[i]
       >= 0
    decreases n - i
    {
        havoc i {hint2}
        use hint2(-1)
        use hint2(1)

        i := i + 1
    }
}
```

[3] https://github.com/anqili426/hhl_frontend

Chapter 5

Visual Studio Code Extension

As in the previous sections, the main goal of this thesis was to improve the usability of Hypra. In its current stage, Hypra is much more than a simple verification tool; it resembles many features of a modern programming language, but its usage is limited. The current implementation as an executable does not support many features developers are accustomed to, such as code highlighting, code auto-completion, and a visual interface. It was therefore natural to explore the creation of an extension for an editor or Integrated Development Environment (IDE).

In this chapter, we present a Visual Studio Code extension that aids development in Hypra by providing features commonly found in extensions of other state-of-the-art verifiers like Viper [11]. It supports most steps during the development process, ranging from code creation to execution and debugging. The supported features include syntax highlighting, auto-completion, snippets (predefined code lines that allow faster writing), and diagnostics (visual error-handling). Furthermore, we wrap the entire functionality of Hypra in the extension so that installation, setup, execution, and modifications can all be done in one place.

We begin by discussing the advantages and disadvantages of a Visual Studio Code extension in Section 5.1. All supported features are presented in Section 5.2. Since this extension will be open for development in the future, we provide an overview of the design and structure of our implementation in Section 5.3.

An example of working with our extension is shown in Figure 5.1, and the extension can be downloaded from Microsoft's Visual Studio Marketplace[1].

[1] https://marketplace.visualstudio.com/items/?itemName=PaulWinkler.hypra-support

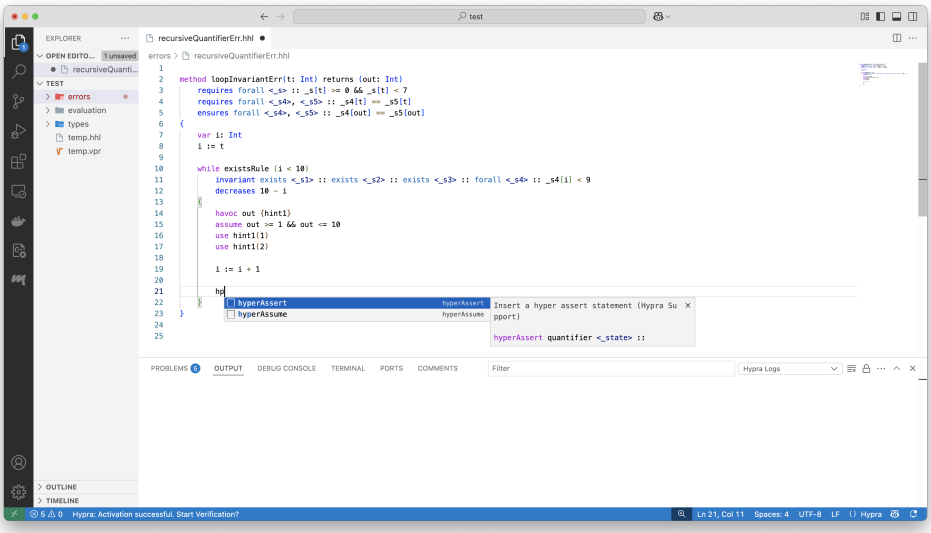

Figure 5.1: Visual Studio Code with the Hypra Support Extension installed.

5.1 Reasoning for a Visual Studio Code Extension

We start this chapter by providing reasoning for our approach to developing an extension for Visual Studio Code, among other possibilities, such as using the Language Server Protocol. Let us thus compare the options.

An extension is a plugin that enhances the capabilities of Visual Studio Code by adding language-specific features, among others. It is editor-specific and can only be used within Visual Studio Code's environment. Furthermore, it allows modifying the visual appearance of the editor, accessing its configurations, and executing other programs.

In contrast, Language Servers are designed to provide language features across multiple editors through a standardized protocol. The supported features often allow for more sophisticated analysis than an extension, but they are restricted to the language itself and cannot modify the editor.

Naturally, most programming languages combine both approaches. However, we decided against this for the following reasons. A language server is a complex system that, in many aspects, mirrors the functionality of a compiler. Developing it is considered challenging and requires thorough planning and testing. Therefore, creating such a system would require a significant amount of engineering [9].

Many of the features we aimed for could also be implemented using solely Visual Studio Code's extension API. Well-written documentation supports the API, and its implementation generally requires less engineering, as many

basic components are already provided. Thus, using the extension API significantly reduces the required engineering effort. As a consequence, the extension's diagnostics typically offer a less sophisticated level of analysis, and in many cases, it is not or only partially supported. However, Hypra already performs diagnostics on its input and thus supports a similar level of analysis as a language server.

Although Hypra was not designed to be run as a language server, we can still simulate real-time diagnostics through a manual execution of Hypra, which works well enough for our purposes. Due to the same reason, non-static auto-completion (recommending options based on the local context) is also not supported. While this feature would be beneficial, it is not strictly necessary. Visual Studio Code offers a simple, static auto-completion for all supported languages, and so these features are still available by creating a language extension for Hypra.

Although the dependence on Visual Studio Code is a drawback, the advantages of easier development for the extension make it a superior alternative in the scope of this thesis. Additionally, with a market share of 73.6%, most users should be able to install our extension [14]. In summary, using a Visual Studio Code extension provides most of our desired features while minimizing the development effort.

5.2 Provided Features

Let us now look at the provided features in more detail. For each one, we will explain the reasoning behind its development and how it enhances the user's experience. We will also provide examples where necessary. More details about the implementation of these features can be found in Section 5.3.

5.2.1 Installation

Our first provided feature is a swift and user-friendly installation of Hypra on the user's local system. The extension automatically installs Hypra and performs the necessary setup, allowing users to start using Hypra immediately after installing the extension. This improves upon our previous installation process, which required manually building the executable and configuring the environment.

Furthermore, updates can be distributed swiftly to users, as the instance contained in the extension can be modified and updated by simply releasing a new version of the extension. Once the extension is installed, Hypra can be executed within the extension, assuming the system meets the required

prerequisites. To avoid restricting Hypra to a single version, users can link custom instances of Hypra to the extension through its settings.

5.2.2 Execution

The extension also improves the execution of Hypra on an input file by streamlining this process and embedding it into Visual Studio Code's graphical interface. Previously, verification was done by executing Hypra with the path to the input file and various configurations as arguments. Our extension enables automatic execution by providing elements in the user interface and editor events to trigger verification. The editor's file is selected as the input, and the configuration is applied automatically. This configuration can also be modified via the extension's settings.

Furthermore, we emulate automatic linting by running Hypra in the background whenever a file is saved. The results of Hypra's execution are displayed in real time within the editor, with errors or successful verification triggering graphical feedback. This creates the appearance of immediate feedback on the user's code during the development process. Alternatively, execution can be triggered by running a Visual Studio Code command, which performs the same operations as before. This process is illustrated in Figure 5.2.

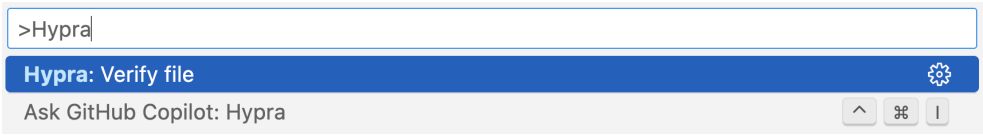

Figure 5.2: Hypra's verification command.

5.2.3 Syntax Highlighting

Our extension also provides syntax highlighting for Hypra's custom programming language. Although not strictly necessary for execution, it is a valuable addition that aligns with current industry standards for programming languages. Furthermore, it aids development by enhancing code readability and comprehension. An example can be seen in Figure 5.3.

5.2.4 Auto-Completion

As previously mentioned, Hypra does not support full auto-completion. We mitigate this by relying on snippets and Visual Studio Code's built-in static auto-completion. The former refers to a set of commonly used code blocks that can be easily inserted into a file by typing a trigger keyword. For example, consider a scenario where a user wants to create a new while loop.

```
/**
* This method finds the maximum element in a sequence of integers.
* It takes a sequence and its length as input and returns the maximum element.
*/
method findMaxInSeq(seq: Seq[Int], l: Int) returns (res: Int)
// precondition
requires forall <_s> :: _s[seq] != Seq[Int]() && _s[l] == |_s[seq]|
// postconditions
ensures forall <_s> :: forall _i: Int :: (_i >= 0 && _i < _s[l]) ==> ((_s[seq])[_i] <= _s[res])
{
    var max: Int
    max := seq[0]

    var i: Int
    i := 1

    while (i < l)
    invariant forall <_s> :: forall _j: Int :: ((_j >= 0 && _j < _s[i]) ==> _s[seq][_j] <= _s[max])
    invariant forall <_s> :: 0 <= _s[i] && _s[i] <= _s[l]
    decreases l - i
    {
        if (seq[i] > max) {
            max := seq[i]
        }
        i := i + 1
    }

    res := max
}
```

Figure 5.3: Hypra's syntax highlighting

They start by writing while and triggering the auto-completion, as shown in Figure 5.4. They can then choose the while loop blueprint that fits their use case, which will be inserted automatically. This helps new users become familiar with the syntax and saves time during development.

The latter refers to a feature provided by Visual Studio Code for all supported languages. The editor automatically remembers tokens declared earlier in the file. If a user types text similar to these tokens, auto-completion is triggered. However, this feature does not account for the language's syntax or the token's semantics and is therefore limited.

5.2.5 Error Handling

This is the most relevant feature our extension provides. As mentioned in Section 5.1, we cannot provide diagnostics during code creation. However, we can offer detailed feedback on errors after running the local Hypra instance. The instance and the extension are closely interlinked, allowing us to send error information from the instance to the extension. This information is then analyzed and translated into graphical feedback within the editor. As a result, an error produces behavior similar to what users are accustomed to in other programming languages.

For example, consider the scenario shown in Figure 5.5. The code contains a

Figure 5.4: A Hydra snippet suggesting the auto-completion of a while loop.

verification error that the user might not immediately notice. As soon as they save the Hydra file, the local instance will—among other things—attempt to verify it. This process will detect the verification error. The extension then receives detailed information about the error and triggers the graphical feedback shown in the figure.

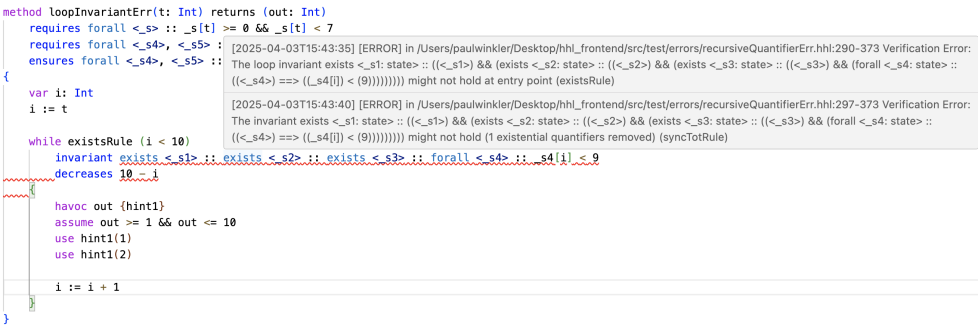

Figure 5.5: A Hypra file containing a program with unsatisfactory specifications. This results in an error message which is shown to the user.

5.3 Design and Implementation

The following section covers the most important aspects of the extension's implementation. Our goal is to equip future developers with the knowledge needed to continue its development. The implementation is based on the Visual Studio Code Extension API and TextMate grammars, which are discussed in more detail in Subsection 5.3.1.

The project is based on the structure proposed by the Extension API and follows the schema presented in Figure 5.6. In most respects, it resembles the structure of a standard Visual Studio Code extension, as seen in various

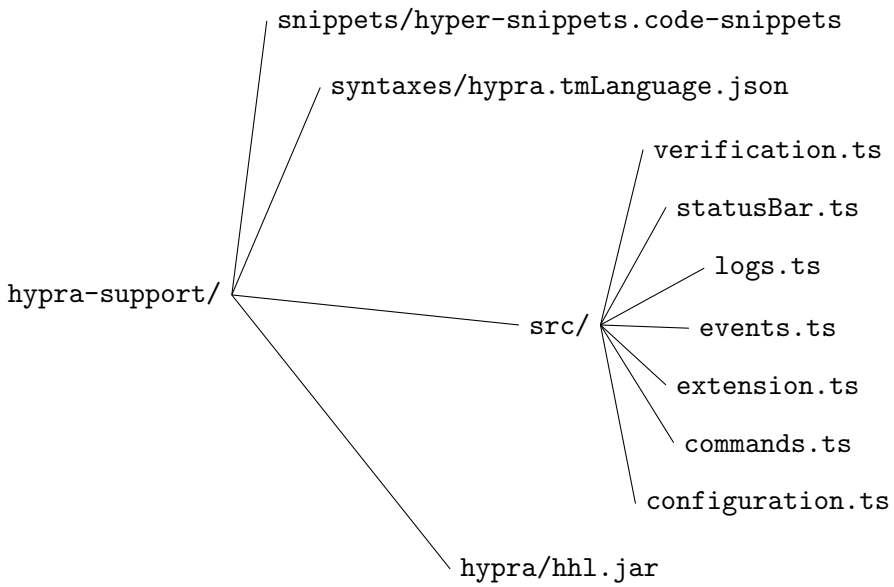

Figure 5.6: The simplified structure of our extension's code-base.

examples available online. Subsections 5.3.2–5.3.5 will present the essential components.

The complete code base is available on GitHub.[2]

5.3.1 Design Choices

As previously mentioned, our entire extension is based on the Visual Studio Code Extension API [10]. It offers a comprehensive package with options for implementing a wide variety of features. Moreover, it is standardized, widely accepted, and supported by many examples and tutorials. For each of the previously mentioned features, its documentation offers a designated page explaining the structure and implementation strategy. We closely followed these official guidelines, and reading the documentation should provide a solid understanding.

We relied on TextMate grammars for defining the syntax highlighting. Again, we followed industry standards to the best of our ability.

For the communication between the extension and the included Hypra instance, we used standard JSON object serialization. The objects were transferred via standard input and output channels. Efficient decoding was enabled by extending the standards created for error messages in Chapter 4 to include general logging messages. The entire standard can be found in the appendix.

[2] https://github.com/pawinkler/hypra-support

5.3.2 Syntaxes

The `syntaxes/` folder contains all files related to the syntax highlighting of our language. As mentioned previously, Hypra's syntax was defined using TextMate grammar in JSON format. The grammar itself is not particularly complex. It works by registering specific keywords used in Hypra and assigning them a class, which in turn determines their color based on the user's installed theme.

5.3.3 Snippets

Snippets are a similar case. Their definitions can be found in the `snippets/` folder, which contains a single file describing all supported snippets. We aimed to include a snippet for every common use case, though additions may be necessary in the future.

5.3.4 Hypra

This is the location of the packaged Hypra instance. The `hypra/` folder contains a pre-built version of the Hypra verifier, which is executed by the extension during the verification process.

Any Hypra instance can be used here, as there is no special version required for the extension. However, some minor modifications to Hypra were necessary. These include changes to the logging system—required to enable efficient data transfer between the extension and the instance—and adjustments to the error handling in Hypra's code base. Any Hypra instance can be run in extension mode by providing the `--ext` flag.

Let us elaborate further on how Hypra is used to simulate a language server. As mentioned in Section 4.2, Hypra essentially acts as a source-to-source compiler. It checks each provided program for syntactic and semantic correctness and provides feedback in case of errors. We make use of this behavior by triggering it when a file is saved. If an error occurs, the information is presented visually and effectively fulfills the same role a language server would provide.

During the development of our extension, the work done on Hypra's error messages proved to be beneficial. Since communication between the instance and the extension is crucial, the standardization of logging and error handling, established in Section 4.1, enabled a smooth integration of Hypra into the extension.

5.3.5 Extension

Finally, the implementation of our extension can be found in the `src/` folder. We will quickly cover the most important aspects:

- `configuration.ts`: Handles the loading, storing, and applying of configurations.
- `commands.ts`: Registers the extension's commands.
- `extension.ts`: Registers the extension itself and prepares the environment.
- `events.ts`: Registers the extension's events.
- `logs.ts`: Implements a standardized logging system.
- `statusBar.ts`: Implements a status bar component for VS Code's user interface.
- `logs.ts`: Handles the verification of a file.

A new developer should be able to understand most of these files based on the brief descriptions and the comments provided in the code. Here, we want to emphasize the core of the extension: the verification process.

During a verification event, which is triggered by saving a Hypra file, clicking on the status bar, or executing the extension's verification command, the `verify` function is called. This initiates the following workflow:

1. The prerequisites of the system are checked.
2. The extension's configuration is read.
3. The input arguments for Hypra are constructed based on the configuration and text editor data.
4. Hypra is spawned as a separate process and provided with all necessary information to verify the file.
5. The extension listens on the standard and error outputs.
6. Events received via these channels are handled and responded to.

Communication is handled by transferring standardized JSON objects. We distinguish between different types of objects, such as *Information*, *Warnings*, and *Errors*. Based on their type, the handler processes these objects accordingly.

For example, an error object contains all the necessary information to generate a visual warning in the code. Consequently, when an error is encountered, such a warning is displayed in the user interface.

Chapter 6

Conclusion and Future Work

This final chapter presents our conclusions and discusses possibilities for future work.

6.1 Conclusion

The work we have proposed and conducted during this thesis broadens the possible use cases of Hypra and improves its usability.

Our initial goals for this thesis were:

1. adding support for new types,
2. improving the generated error messages,
3. and implementing a Visual Studio Code Extension for Hypra.

For each goal, we presented the issue underlying the specific goal, proposed a solution, and, finally, modified Hypra's code base to fully support our solution. Additionally, we tested our implementation to ensure correctness.

Hypra now supports a larger class of programs and can verify new properties regarding boolean-typed variables, sequences, sets, and maps. The improved error messages ensure a constructive and straightforward development and debugging process, which covers Hypra-specific challenges, such as the encoding of while loops. All of the previous topics are additionally supported by providing an extension for Visual Studio Code, which offers many additional features to support development using Hypra.

Furthermore, we have tested our proposed solutions and extended Hypra's existing evaluation. Our analysis shows that our modifications indeed improve Hypra's usability while requiring an acceptable runtime and number of proof annotations. Although individual examples have also shown that using composite types can significantly degrade both.

6.2 Future Work

We conclude this thesis by discussing several possibilities for future work related to this project.

6.2.1 Verifying Compatibility with Proof Variables

During the implementation of our new types, some modifications were made to the parser, symbol table, and type checker. Many of the implemented features overlap with the implementation of our proof variables. Proof variables are used in Hypra as placeholders in logical expressions or proofs, often to preserve certain information. Our implementation could not be fully tested for compatibility with the proof variables implementation. Therefore, it is essential for future development to conduct extensive testing in this regard.

6.2.2 Improving Well-Definedness Tests for Sequences and Maps

As described in Section 3.3, our encoding handles operations that could result in runtime errors in a special fashion. These operations are element accesses on Sequences and Maps. If it cannot be proven that an access is well-defined, we map this situation to an error state by asserting the well-definedness check. However, this is not valid for accesses in postconditions. Postconditions are not part of the program and should thus not generate any runtime error. An example of such a program can be seen in Listing 6.1.

We currently treat this situation by assuming illegal access to be underspecified. However, this is not fully consistent with the previous approach and might confuse users. Thus, it might make sense to develop a special error to cover this case. It could be used to inform a user about the situation and would create a standardized procedure.

Listing 6.1: A program exhibiting unknown behavior due to an runtime error in its postcondition. The precondition ensures that the sequence has length five, resulting in an out-of-bounds error in the postcondition.

```
method unkownBehavior(arr: Seq[Int]) returns (out:
    Int)
     requires forall <_s> :: |_s[arr]| == 5
     ensures forall <_s> :: _s[out] == _s[arr][10]
{
     out := arr[4]
}
```

6.2.3 Adding User-Defined Types

Our implementation allows users to use four additional types, but these are fundamentally restricted to types already included in Hypra. It would be beneficial if Hypra could support user-defined types like domains in the closely related Viper verifier. Domains would allow the definition of additional types, including functions and axioms that describe their properties [11]. This would enable the definition of new user-defined types and significantly broaden Hypra's use cases. An example can be seen in Figure 6 in Dardinier et al. [4]. It contains a program that is currently not supported by Hypra, as it requires the XOR operator, which could easily be implemented using custom types.

6.2.4 Developing a Language Server for Hypra

Currently, our extension lacks a language server and emulates this functionality manually or by passing it to Hypra directly. As discussed in Section 5.3, this works well enough for our requirements, but it still has some drawbacks. Since Hypra needs to be called in the background, diagnostics and autocompletion are only available in a limited form. Additionally, the extension is dependent on Visual Studio Code. If Hypra becomes more widely used in the future and its environment expands, it might make sense to develop a language server and integrate it with our extension. Our extension is developed in a modular fashion, so linking the language server should not be too difficult.

Bibliography

[1] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 14–25, New York, NY, USA, 2004. Association for Computing Machinery.

[2] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, CSF '08, page 51–65, USA, 2008. IEEE Computer Society.

[3] Thibault Dardinier, Anqi Li, and Peter Müller. Hypra: A deductive program verifier for hyper hoare logic. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024.

[4] Thibault Dardinier and Peter Müller. Hyper hoare logic: (dis-)proving program hyperproperties. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.

[5] Osman Hasan and Sofiene Tahar. Formal verification methods. In *Encyclopedia of Information Science and Technology, Third Edition*, pages 7162–7170. IGI Global Scientific Publishing, 2015.

[6] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[7] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[8] Matthew Lutze, Philipp Schuster, and Jonathan Immanuel Brachthäuser. The simple essence of monomorphization. *Proc. ACM Program. Lang.*, 9(OOPSLA1), April 2025.

[9] Microsoft. *Language Server Protocol Specification*, 2024. Accessed: 2024-04-03.

[10] Microsoft. *Visual Studio Code Extension API*, 2024. Accessed: 2024-04-03.

[11] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[12] Peter W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[13] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–69, 2016.

[14] Stack Overflow. Stack Overflow Developer Survey 2024, 2024. Accessed: 2024-04-03.

[15] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in f*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 256–270, New York, NY, USA, 2016. Association for Computing Machinery.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. **In consultation with the supervisor**, one of the following two options must be selected:

☒ I hereby declare that I authored the work in question independently, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

☐ I hereby declare that I authored the work in question independently. In doing so I only used the authorised aids, which included suggestions from the supervisor regarding language and content and generative artificial intelligence technologies. The use of the latter and the respective source declarations proceeded in consultation with the supervisor.

Title of paper or thesis:

Improving a Deductive Program Verifier for Hyperproperties

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Winkler	Paul Maria

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guidelines.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

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

Place, date	**Signature(s)**
~~Trond~~ Trondheim, Norway, 24.04.2025	P. Winkler

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] For further information please consult the ETH Zurich websites, e.g. https://ethz.ch/en/the-eth-zurich/education/ai-in-education.html and https://library.ethz.ch/en/researching-and-publishing/scientific-writing-at-eth-zurich.html (subject to change).