Hypra: A Deductive Program Verifier for Hyper Hoare Logic

THIBAULT DARDINIER*, ETH Zurich, Switzerland
ANQI LI*, ETH Zurich, Switzerland
PETER MÜLLER, ETH Zurich, Switzerland

Hyperproperties relate multiple executions of a program and are useful to express common correctness properties (such as determinism) and security properties (such as non-interference). While there are a number of powerful program logics for the deductive verification of hyperproperties, their automation falls behind. Most existing deductive verification tools are limited to safety properties, but cannot reason about the existence of executions, for instance, to prove the violation of a safety property. Others support more flexible hyperproperties such as generalized non-interference, but have limitations in terms of the programs and proof structures they support.

In this paper, we present the first deductive verification technique for arbitrary hyperproperties over multiple executions of the same program. Our technique automates the generation of verification conditions for Hyper Hoare Logic. Our key insight is that arbitrary hyperproperties and the corresponding proof rules can be encoded into a standard intermediate verification language by representing *sets of states* of the input program *explicitly* in the states of the intermediate program. Verification is then automated using an existing SMT-based verifier for the intermediate language. We implement our technique in a tool called Hypra and demonstrate that it can reliably verify complex hyperproperties.

CCS Concepts: • **Theory of computation → Logic and verification**; **Automated reasoning**; **Hoare logic**.

Additional Key Words and Phrases: Hyperproperties, Deductive Verification, Incorrectness Logic

ACM Reference Format:
Thibault Dardinier, Anqi Li, and Peter Müller. 2024. Hypra: A Deductive Program Verifier for Hyper Hoare Logic. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 316 (October 2024), 30 pages. https://doi.org/10.1145/3689756

1 Introduction

Hyperproperties [Clarkson and Schneider 2008] relate multiple executions of a program. They can be used to express correctness properties such as determinism (two executions of a program with the same input will result in the same output) or monotonicity (executing the program with a larger input will result in a larger output). Hyperproperties are also used to express security and information-flow properties, such as non-interference.

There are numerous program logics that enable the formal verification of different kinds of hyperproperties. Relational Hoare Logic (RHL) [Benton 2004] is able to to reason about 2-safety hyperproperties, that is, properties that should hold *for all pairs of executions* of the same program (such as determinism). Since these properties relate all possible "first" executions to all possible "second" executions, we call them ∀∀*-properties*. RHL has then been extended to support k-safety hyperproperties [Sousa and Dillig 2016], i.e., properties that should hold *for all combinations of k*

*These authors contributed equally to this work.

Authors' Contact Information: Thibault Dardinier, ETH Zurich, Department of Computer Science, Zürich, Switzerland, thibault.dardinier@inf.ethz.ch; Anqi Li, ETH Zurich, Department of Computer Science, Zürich, Switzerland, anqi.li@inf.ethz.ch; Peter Müller, ETH Zurich, Department of Computer Science, Zürich, Switzerland, peter.mueller@inf.ethz.ch.

ACM 2475-1421/2024/10-ART316
https://doi.org/10.1145/3689756

executions of the same program ($\forall^k$-properties), such as transitivity ($k = 3$) or associativity ($k = 4$). We call program logics for safety hyperproperties (i.e., $\forall^*$-properties) *overapproximate*, because they work by overapproximating the set of possible executions.

Many important hyperproperties fall outside the class of k-safety hyperproperties, because they require proving the *existence* of executions. For the special case of one execution, Reverse Hoare Logic [de Vries and Koutavas 2011] and Incorrectness Logic [O'Hearn 2019] allow one to prove the existence of an execution, for instance, the reachability of a bug. We call such properties *$\exists$-properties*; the corresponding logics *underapproximate* the set of possible executions.

More complex hyperproperties, of the form $\forall^*\exists^*$ or $\exists^*\forall^*$, require program logics that can perform both overapproximation and underapproximation reasoning. An important example is *Generalized Non-Interference* (GNI) [McCullough 1987; McLean 1996], a $\forall\forall\exists$-hyperproperty. GNI requires, for any two executions τ_1 and τ_2 with the same low-sensitivity inputs, the existence of a third execution with the same high inputs as τ_1 and the same low output as τ_2. Examples of $\exists^*\forall^*$-hyperproperties include the violation of GNI ($\exists\exists\forall$) and the existence of an execution with minimal values ($\exists\forall$). Several recent program logics support these more-complex hyperproperties. RHLE [Dickerson et al. 2022] supports $\forall^*\exists^*$-hyperproperties, BiKAT [Antonopoulos et al. 2023] supports $\forall\exists$-hyperproperties and (in principle) $\exists\forall$-hyperproperties, and Hyper Hoare Logic [Dardinier and Müller 2024] supports hyperproperties with arbitrary quantifier alternations, including both $\forall^*\exists^*$ and $\exists^*\forall^*$-hyperproperties.

Automation. Verification in program logics for safety properties of single executions (such as Hoare Logic) can be automated using *deductive program verifiers* (*verifiers* for short), such as Boogie [Leino 2008], Dafny [Leino 2010], Viper [Müller et al. 2016], or Why3 [Filliâtre and Paskevich 2013]. Given a program, a specification, and hints from the user (such as loop invariants), these tools attempt to prove that the program satisfies the specification by computing proof obligations and solving them using an SMT solver.

Deductive verifiers for hyperproperties are mostly limited to k-safety hyperproperties, which can be reduced to safety properties for a product program [Barthe et al. 2011; Eilers et al. 2019] and then verified using an off-the-shelf verifier. Alternatively, there are dedicated verifiers for hyperproperties, such as WhyRel [Nagasamudram et al. 2023], SecC (based on SecCSL [Ernst and Murray 2019]), and HyperViper (based on CommCSL [Eilers et al. 2023]) for 2-safety hyperproperties, and Descartes (based on Cartesian Hoare Logic [Sousa and Dillig 2016]) for k-safety hyperproperties.

ORHLE (based on RHLE [Dickerson et al. 2022]) is the only deductive verifier that goes beyond k-safety hyperproperties by supporting $\forall^*\exists^*$-hyperproperties. However, it is limited to a *fixed* quantification scheme; users have to first fix the numbers of $\forall$-quantifiers and $\exists$-quantifiers and then write preconditions, postconditions, and loop invariants in this fixed scheme. It is, thus, not possible to compose proofs with different quantification schemes, e.g., to use a $\forall$-property in the proof of a $\forall\forall$-property. Moreover, ORHLE supports relational loop invariants only when the different executions perform the same number of loop iterations, which limits the programs and hyperproperties that can be verified in practice.

To the best of our knowledge, no existing verifier supports properties beyond $\forall^*\exists^*$-hyperproperties, such as $\exists^*\forall^*$-hyperproperties.

This work. We present the first deductive verifier for hyperproperties with arbitrary quantifier alternations. Our tool, Hypra, is based on Hyper Hoare Logic (HHL) [Dardinier and Müller 2024]. Unlike CHL and RHLE, which have dedicated verifiers, HHL had lacked a verifier until Hypra was developed. Like HHL, Hypra allows assertions to quantify explicitly over states, giving users the flexibility to express and combine different types of hyperproperties in the same proof. Going

beyond HHL, Hypra supports reasoning about runtime errors (e.g., to prove the existence or absence of bugs).

Our key insight is that arbitrary hyperproperties and the corresponding proof rules can be encoded into a standard intermediate verification language by representing *sets of states* of the input program *explicitly* in the states of the intermediate program. Verification is then automated using an existing SMT-based verifier for the intermediate language. To ensure that SMT solvers can handle the resulting verification conditions, our encoding carefully manages quantifier instantiations by tracking simultaneously a lower bound and an upper bound of the sets of states. Note that, in contrast to product constructions, our encoding does *not* duplicate the statements of the input program and can handle arbitrary hyperproperties (beyond k-safety).

Like HHL, we focus on hyperproperties that relate multiple executions of the *same* program; relating executions of *different* programs (e.g., to prove their equivalence) poses additional challenges (such as finding an alignment), which are orthogonal to the problems addressed here. Both WhyRel and ORHLE support such relational proofs.

Contributions. In summary, our work makes the following contributions:

(1) We extend Hyper Hoare Logic [Dardinier and Müller 2024] with the ability to reason about runtime errors. This allows us to prove correctness (the absence of bugs), incorrectness (the existence of bugs) and more complex hyperproperties (e.g., proving that the occurrence of a runtime error does not leak any secret information).
(2) We present the first approach to generate verification conditions for hyperproperties with arbitrary quantifier alternations for loop-free statements. The resulting verification conditions are amenable to SMT solvers.
(3) HHL provides multiple loop rules for different kinds of programs and properties. We present our approach to automatically select which loop rule to apply, such that users are not exposed to the details of the underlying logic. This automatic selection is important when dealing with $\exists^*\forall^*$ loop invariants, which require the application of several different loop rules. Moreover, we present and prove sound a new loop rule for $\forall^*\exists^*$-hyperproperties, which is easier to automate than the corresponding rule in HHL.
(4) We implement our approach in a tool called Hypra, based on the Viper intermediate language [Müller et al. 2016]. Our evaluation on a set of benchmarks from the literature shows that Hypra can prove a large class of hyperproperties for a large class of programs, in a reasonable amount of time and with a reasonable amount of proof annotations.

Outline. The rest of the paper is organized as follows: Sect. 2 highlights the capabilities of our verifier through several examples, and presents our extension of Hyper Hoare Logic to reason about runtime errors. Sect. 3 presents our approach to generate verification conditions for loop-free statements, while Sect. 4 handles loops. We discuss the implementation and evaluation of Hypra in Sect. 5, related work in Sect. 6, and limitations and future work in Sect. 7.

2 A Tour of the Verifier

In this section, we illustrate the key capabilities of our verifier on several examples. Sect. 2.1 shows how Hypra can perform both over- and underapproximation, and how combining both allows us to prove complex hyperproperties (such as $\exists^*\forall^*$-hyperproperties). Sect. 2.2 illustrates how Hypra reasons about runtime errors, in particular how it can prove the absence of errors, the existence of errors, and more complex (hyper)properties (such as the fact that the occurrence of a runtime error does not leak secret information). We also explain how we extend Hyper Hoare Logic, which does

```
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)
```

Fig. 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.

```
method randNat() returns (y: Int)
    requires ∃⟨σ⟩. ⊤
    ensures ∀⟨σ⟩. 1 ≤ σ(y) ≤ 2
    ensures ∃⟨σ⟩. σ(y) = 1
    ensures ∃⟨σ⟩. σ(y) = 2
{
    var x: Int
    x := nonDet() // {hint}
    // use hint(0,1)
    if (x > 0) {
        y := 1
    }
    else {
        y := 2
    }
}
```

```
method secure(h: Int, l: Int) returns (o: Int)
    requires ∀⟨σ1⟩, ⟨σ2⟩. σ1(l) = σ2(l)
    ensures ∀⟨σ1⟩, ⟨σ2⟩. σ1(o) = σ2(o)
{
    if (h > 0) { o := 2 * l }
    else { o := l }
    if (h <= 0) { o := o + l }
}
```

```
method leaky(h: Int) returns (o: Int)
    requires ∃⟨σ1⟩, ⟨σ2⟩. σ1(h) < σ2(h)
    ensures ∃⟨σ1⟩, ⟨σ2⟩. ∀⟨σ⟩. σ(o)=σ1(o)
                                ⇒σ(h)≠σ2(h)
{
    var y: Int
    y := nonDet() // {hint}
    assume 0 <= y <= 10
    // use hint(10)
    o := h + y
}
```

Fig. 2. Examples of overapproximation, underapproximation, and hyperproperties. The example on the left illustrates ∀-properties and ∃-properties of individual program executions. The examples on the right illustrate hyperproperties. Method `secure` satisfies non-interference, a ∀∀-hyperproperty. Method `leaky` (which is adapted from Dardinier and Müller [2024]) violates generalized non-interference. The negation of this property is expressed as an ∃∃∀-hyperproperty. All examples are successfully verified by Hypra, using the provided hints to construct witnesses for existential quantifiers.

not support errors, to do so. Finally, Sect. 2.3 shows how Hypra handles while loops. All examples shown in this section are successfully and automatically verified by our tool.

2.1 Overapproximation, Underapproximation, and Hyperproperties

Hypra establishes *hyper-triples* of the form $[P]\ C\ [Q]$, where C is a program statement, and P (the precondition) and Q (the postcondition) are *hyper-assertions*, i.e., predicates over sets of states (which we formally define in Sect. 2.2). Intuitively, the triple $[P]\ C\ [Q]$ means that for any set S of initial states that satisfies the precondition P, the set S' of all states (including error states) reachable by executing C from any state in S satisfies the postcondition Q.

Input language. Hypra supports the input language shown in Fig. 1. In addition to the syntax shown in the figure, non-deterministic assignments can be annotated with hint declarations to guide the automatic verification (as we explain in the next subsection). To use a declared hint, users can write annotations with the **use** keyword. In its current version, Hypra supports integer program variables and the usual arithmetic operations.

Over- and underapproximation. As an example, consider the method `randNat` in Fig. 2 (left). This method non-deterministically chooses a value for x (which can for example represent a random choice or some user input), and assigns 1 to y if $x > 0$, and 2 otherwise. In other words, this method non-deterministically returns 1 or 2. The keyword **requires** specifies the precondition of the method, while the keyword **ensures** specifies the postcondition. Multiple preconditions or postconditions are simply conjoined. Thus, this example corresponds to the hyper-triple

$$[\exists\langle\sigma\rangle.\,\top]\ C_r\ [(\forall\langle\sigma\rangle.\,1 \leq \sigma(y) \leq 2) \wedge (\exists\langle\sigma\rangle.\,\sigma(y) = 1) \wedge (\exists\langle\sigma\rangle.\,\sigma(y) = 2)]$$

where C_r refers to the body of method `randNat`. Let S' be the set of reachable states at the end of the method. The postcondition $\forall\langle\sigma\rangle.\,1 \leq \sigma(y) \leq 2$, which is equivalent to $\forall\sigma \in S'.\,1 \leq \sigma(y) \leq 2$, *overapproximates* the set S'; It means that, in any final state (from S'), y will either be 1 or 2, corresponding to the standard Hoare triple $\{\top\}\ C_r\ \{1 \leq y \leq 2\}$. On the other hand, the postconditions $\exists\langle\sigma\rangle.\,\sigma(y) = 1$ and $\exists\langle\sigma\rangle.\,\sigma(y) = 2$, equivalent to $\exists\sigma \in S'.\,\sigma(y) = 1$ and $\exists\sigma \in S'.\,\sigma(y) = 2$, respectively, *underapproximate* the set S': They express the existence of two reachable final states with $y = 1$ and $y = 2$. Conjoined, these three postconditions express that this method has only two possible outcomes for y, 1 and 2, and that both outcomes are reachable. The precondition $\exists\langle\sigma\rangle.\,\top$ expresses that the set of initial states is non-empty. This precondition is required for the hyper-triple to hold, otherwise the postconditions $\exists\langle\sigma\rangle.\,\sigma(y) = 1$ and $\exists\langle\sigma\rangle.\,\sigma(y) = 2$ would not hold (because no states are reachable from an empty set of initial states).

While the first postcondition verifies automatically, proving the existentially-quantified post-conditions requires a user-provided hint. *Hints* are annotations for non-deterministic assignments that give examples of values that might be assigned. Hypra uses this information to construct witness states for $\exists$-properties. In our example, the `{hint}` annotation after the non-deterministic assignment introduces an identifier for this assignment, and the annotation **use** `hint(0,1)` tells the verifier that 0 and 1 are relevant choices for this non-deterministic assignment (technically, hints are used to provide triggers for the quantifier instantiation in the SMT solver).[1] The two values ensure that both branches of the subsequent conditional statement are considered, which is necessary to prove both existentially-quantified postconditions. The specific values are irrelevant in this example; any pair of a non-negative and a positive integer would work.

Hyperproperties. Overapproximation allows us to formally verify safety hyperproperties such as non-interference, as illustrated by method `secure` in Fig. 2 (top right). The specification expresses that the output o depends only on the low-sensitive input l and, thus, does not leak any information about the secret input h. Hypra verifies this example without further annotations.

By enabling both overapproximation and underapproximation reasoning, our approach can verify more complex hyperproperties, such as $\forall^*\exists^*$-hyperproperties or $\exists^*\forall^*$-hyperproperties, as illustrated by the method `leaky` in Fig. 2 (bottom right). The statements `y := nonDet()` and **assume** `0 <= y <= 10` model a non-deterministic choice between 0 and 10. This method leaks information about its secret input h via its output o: h is between $o - 10$ and o. To prove this claim, we formally verify that the method violates generalized non-interference [McCullough 1987;

[1]Note that, in general, hints can depend on the variables of one or more states. For example, **use** $\forall\langle\sigma\rangle.\,\texttt{hint}(\sigma(n))$ tells the SMT solver to consider the value of variable n in all relevant states σ. Hints can also depend on multiple quantified states, as in **use** $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\,\texttt{hint}(\sigma_1(a) + \sigma_2(b))$.

```
method buggy() returns (x: Int)
    requires ∃⟨σ⟩. ⊤
    ensures ∃⟨σ⟩_er. σ(x) = 1
    ensures ∃⟨σ⟩_er. σ(x) = 2
{
    x := randNat()
    var y: Int := x + x
    assert y % 2 == 1
}
```

```
method almostCorrect(x:Int) returns (o:Int)
    requires ∀⟨σ⟩. σ(x) ≥ 0
    ensures ∀⟨σ⟩_er. σ(x) = 0 ∧ σ(o) = 0
{
    o := nonDet()
    assume o >= 0
    var y: Int := x + o
    assert y > 0
}
```

```
method lowError(h: Int, l: Int, t: Int)
    requires ∃⟨σ1⟩, ⟨σ2⟩. σ1(t) = 1 ∧ σ2(t) = 2
    requires ∀⟨σ1⟩, ⟨σ2⟩. σ1(l) = σ2(l)
    ensures (∃⟨σ1⟩_er. σ1(t) = 1) ⇔ (∃⟨σ2⟩_er. σ2(t) = 2)
{
    if (h > 0) {
        assert l >= 0
    }
    if (l < 0) {
        assert false
    }
}
```

Fig. 3. Reasoning about runtime errors. In the top left example, the postconditions specify two possible executions that lead to a runtime error (an assertion violation). The postcondition on the top right expresses that the method fails *only if* both `x` and `o` are 0. The example on the bottom illustrates reasoning about runtime errors in the context of hyperproperties. The specification expresses that the occurrence of a runtime error does not depend on the secret input `h`. All examples are successfully verified by Hypra without any hints.

Volpano et al. 1996]. That is, we prove the existence of two executions with distinct secret values for h that can be *distinguished*. The postcondition expresses that observing the output $\sigma_1(o)$ rules out the possibility that the secret value of h was $\sigma_2(h)$, thus leaking information about the initial value of h. Verifying this existentially-quantified postcondition requires a hint; choosing the provided value 10 for $\sigma_2(y)$ yields the required witnesses.

2.2 Reasoning about Runtime Errors

Our examples so far reason about properties of *normal states*, that is, states that are reached when the program executes successfully. In addition, our technique can also reason about a set of *error states*, which are reached when a runtime error occurs. This feature allows us to prove both the absence and presence of runtime errors, as well as advanced hyperproperties such as failure-sensitive non-interference.

Method `buggy` in Fig. 3 (top left) calls method `randNat` (see Fig. 2), doubles the result, and asserts that the resulting value is odd. The postconditions express the existence of two failing executions, à la Incorrectness Logic [O'Hearn 2019]: There exist error states σ where the result of `randNat` is 1 and 2, respectively.

The ability to quantify over error states allows us to express more complex properties. For example, the specification of method `almostCorrect` in Fig. 3 (top right) expresses that a runtime

error occurs *only if* the non-deterministic value assigned to o is 0 *and* the input x is 0. This *almost-correctness* is captured by the universal quantification in the postcondition which expresses that all error states satisfy $x = 0 \land o = 0$, that is, no other execution fails.

The absence of errors can be specified via the postcondition $\forall\langle\sigma\rangle_{\text{er}}.\ \bot$, which expresses that the set of error states is empty.

In the context of hyperproperties, reasoning about error states is, for instance, useful to express failure-sensitive non-interference, that is, the fact that observing a runtime error does not leak secret information. The specification of method `lowError` in Fig. 3 (bottom) expresses this property. For two different executions with the same value for the low-sensitive input l (but potentially different values for the secret input h), one execution fails if and only if the other execution fails. In particular, this proves that the occurrence of a runtime error does not depend on h, such that observing an error does not leak secret information. In this specification, the parameter t is used to *tag* executions, which allows us to identify the pre- and post-state of a given execution. Tags are expressed as logical variables in Hyper Hoare Logic, but represented by (immutable) program variables in Hypra.

Extending Hyper Hoare Logic to support runtime errors. We have extended Hyper Hoare Logic, which does not provide any support for reasoning about errors, as follows:

Definition 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 $sem(C, S) \triangleq \{\sigma' \mid \exists\sigma \in S.\ \langle C, \sigma\rangle \to \sigma'\}$, while the set of* error states *reachable by executing C in any state from S is denoted as $err(C, S) \triangleq \{\sigma' \mid \exists\sigma \in S.\ \langle C, \sigma\rangle_{er} \to \sigma'\}$.*

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:*

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

Note that we start with an *empty* set of error states (second argument of P) to distinguish clearly between the errors caused by a statement C and those caused by statements executed prior to C. In particular, for a sequential composition $C_1; C_2$, the set of error states that come from C_2 depends on $sem(C_1, S)$ only, but not on $err(C_1, S)$; formally, $err(C_1; C_2, S) = err(C_1, S) \cup err(C_2, sem(C_1, S))$. Consequently, prescribing a specific set of error states in C_2's precondition is not useful.

Specification language. Hypra supports the following syntax for hyper-assertions P where E ranges over integer expressions, B over Boolean expressions, and P over hyper-assertions:

$$E ::= \sigma(y) \mid x \mid n \mid E + E \mid E - E \mid E * E \mid E\ /\ E \mid E\ \%\ E \mid \ldots$$
$$B ::= \top \mid \bot \mid E = E \mid E > E \mid E \geq E \mid \neg B \mid \ldots$$
$$P ::= \forall\langle\sigma\rangle.\ P \mid \exists\langle\sigma\rangle.\ P \mid \forall\langle\sigma\rangle_{\text{er}}.\ P \mid \exists\langle\sigma\rangle_{\text{er}}.\ P \mid \forall x.\ P \mid \exists x.\ P \mid P \land P \mid P \lor P \mid P \Rightarrow P \mid B$$

Hyper-assertions interact with the set of normal states through the quantifiers $\forall\langle\sigma\rangle$ and $\exists\langle\sigma\rangle$, and with the set of error states via the quantifiers $\forall\langle\sigma\rangle_{\text{er}}$ and $\exists\langle\sigma\rangle_{\text{er}}$. The quantifiers $\forall\langle\sigma\rangle_{\text{er}}$ and $\exists\langle\sigma\rangle_{\text{er}}$ are not allowed in preconditions (since we always start with an empty set of error states).

```
method minimum(n: Int) returns (x: Int, y: Int)
    requires ∃⟨σ⟩. ⊤
    requires ∀⟨σ1⟩, ⟨σ2⟩. σ1(n) = σ2(n)
    ensures ∃⟨σ⟩. ∀⟨σ'⟩. σ(x) ≤ σ'(x) ∧ σ(y) ≤ σ'(y)
{
    var i, r: Int
    i, x, y := 0
    while (i < n)
        invariant ∀⟨σ1⟩, ⟨σ2⟩. σ1(n) = σ2(n) ∧ σ1(i) = σ2(i)
        invariant ∃⟨σ⟩. ∀⟨σ'⟩. σ(x) ≤ σ'(x) ∧ σ(y) ≤ σ'(y)
        decreases n - i
    {
        r := nonDet() // {hint}
        assume r >= 5
        // use hint(5)
        x := x + y + 2 * i + 3 * r
        y := x + 3 * i + 2 * r
        if (x >= n) { y := y + r }
        i := i + 1
    }
}
```

Fig. 4. Reasoning about loops. Given a loop invariant and an optional variant, Hypra automatically selects the appropriate loop rule. Like all other assertions, loop invariants may express arbitrary hyperproperties, here, the existence of an execution with minimal values for x and y. The example is successfully verified by Hypra.

2.3 Reasoning about Loops

Hyper Hoare Logic provides four different loop rules that can prove different flavors of hyperproperties. These rules are applicable in different contexts; for example, some rules are applicable only if all loop executions perform the same number of iterations, and others only if the loop is proved to terminate. Based on a user-provided loop invariant and an optional loop variant, Hypra determines automatically which rule to apply. This allows users to reason about loops in a familiar way without being exposed to the complexity of the underlying logic, as we illustrate on method `minimum` in Fig. 4.

This method starts with $x = y = 0$ and performs n loop iterations, during which it modifies the values of x and y in a non-deterministic way. We want to prove that, given a fixed value for the input n (enforced by the precondition $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle. \sigma_1(n) = \sigma_2(n)$), there exists an execution where both x and y have minimal values at the end of the method (without specifying their values, which depend on n non-deterministic choices). The proof is based on a user-provided relational *loop invariant*, which must hold before the loop, and after every iteration. The first part of the loop invariant, $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle. \sigma_1(n) = \sigma_2(n) \land \sigma_1(i) = \sigma_2(i)$, ensures that all states have the same values for i and n, and thus that all executions will exit the loop simultaneously. Our verifier automatically detects this pattern, and uses a specialized loop encoding to handle it, as we will explain in Sect. 4. Moreover, knowing that all executions have the same value for i is necessary to prove the existence of an execution with minimal values. The second part of the loop invariant, $\exists\langle\sigma\rangle. \forall\langle\sigma'\rangle. \sigma(x) \leq \sigma'(x) \land \sigma(y) \leq \sigma'(y)$, ensures that, after any number of iterations, there exists an execution with minimal values for x and y, which corresponds to our postcondition. Finally, we

need to prove that the loop terminates, otherwise our postcondition would not hold: If the loop did not terminate, then *no* final state would exist. We prove termination using a standard loop variant (following the **`decreases`** keyword). The verifier checks, for all states, that this loop variant is well-founded (non-negative), and that it strictly decreases during each iteration, thus ensuring termination.

This section illustrated the capabilities of our verification approach from a user's perspective. In the next two sections, we will explain how we compute verification conditions by encoding the input program into an intermediate verification language, for which an automated verifier exists.

3 Verification Conditions for Loop-Free Statements

Given a loop-free program statement C (potentially containing hints), a precondition P and a postcondition Q, our verifier generates a Viper [Müller et al. 2016] program, such that validity of the Viper program implies validity of the hyper-triple $\models [P]\ C\ [Q]$. Our key insight is to represent *sets of states* of the input program C as *single states* in the Viper program. More precisely, the Viper program contains set-valued variables, whose contents represent the set of states. Intuitively, the generated Viper program starts with a set-valued variable S (containing an arbitrary value), assumes that S satisfies the precondition P (via an assume-statement in the Viper program), tracks the sets of normal states and error states that can be reached by executing C in any state from S (by updating the set-valued variable S accordingly), and checks whether they satisfy the postcondition Q (via an assert-statement in the Viper program). To avoid clutter, we often ignore the set of error states in the rest of this paper, and focus only on the set of normal states (which we also call the set of reachable states), but the same principles apply to both.

As we will explain in Sect. 3.2.1, verification conditions that quantify both universally and existentially over the set of states S pose major difficulties for SMT solvers. To avoid this problem, our encoding tracks separately an *upper bound* and a *lower bound* of the set of reachable states. The lower bound is sufficient to verify safety hyperproperties ($\forall^*$-hyperproperties), while the upper bound is sufficient to verify $\exists^*$-hyperproperties. Reasoning about hyperproperties with arbitrary quantifier alternations, such as $\forall^*\exists^*$ or $\exists^*\forall^*$-hyperproperties, requires combining both encodings. However, naively combining the two encodings does not work, as it can lead to matching loops where the SMT solver gets stuck in an infinite instantiation of quantifiers.

In Sect. 3.1, we present our approach to track an upper bound and a lower bound of the set of reachable states. We then explain, in Sect. 3.2, why naively combining both can lead to matching loops, and present our solution, which allows us to reason about hyperproperties with arbitrary quantifier alternations, while avoiding matching loops.

The Viper verification infrastructure. Viper [Müller et al. 2016] is an intermediate verification language (similar to Boogie [Leino 2008]) accompanied by two back-end verifiers, based on symbolic execution and on verification condition generation, respectively, and both back-ends use the underlying SMT solver Z3 [de Moura and Bjørner 2008]. A Viper back-end verifier takes as input a Viper program annotated with method pre- and post-conditions, and verifies each method modularly. It reports a verification success if all methods have been successfully verified, or one or more error messages otherwise. Hypra works with both back-end verifiers. The Viper language provides the standard features of a guarded-command language (assert and assume, together with the usual control structures). Moreover, Viper provides an assertion language based on first-order predicate logic (including quantifiers), and the possibility to extend this language with user-defined background theories, which our technique uses to model (potentially infinite) sets. While Viper has been specifically designed to automate separation logic [Reynolds 2002], Hypra does not use any

separation logic specific features, so the same encoding would work with similar SMT-based tools, such as Boogie.[2]

3.1 Upper and Lower Bounding the Set of Reachable States

Starting from a set of states S satisfying the user-provided precondition, the high-level goal of our Viper encoding is to construct the set of reachable states $sem(C, S)$ (introduced in Def. 1), to check whether the user-provided postcondition holds for the latter. To do so, we construct a lower bound $S_\forall$ and an upper bound $S_\exists$ for $sem(C, S)$, by leveraging the following properties, as we will show below:

$$\forall \sigma'.\, \sigma' \in S_\forall \Rightarrow \exists \sigma.\, \sigma \in S \wedge \langle C, \sigma\rangle \rightarrow \sigma' \tag{1}$$

$$\forall \sigma, \sigma'.\, \sigma \in S \wedge \langle C, \sigma\rangle \rightarrow \sigma' \Rightarrow \sigma' \in S_\exists \tag{2}$$

Property (1) says that every state $\sigma' \in S_\forall$ results from executing C in some state $\sigma \in S$, while property (2) says that any final state σ' that results from executing C in a state $\sigma \in S$ belongs to $S_\exists$. Therefore, properties (1) and (2) imply that $S_\forall \subseteq sem(C, S) \subseteq S_\exists$.

Encoding the postcondition using $S_\forall$ and $S_\exists$. Intuitively, (1) is useful when we have $\sigma' \in S_\forall$ as an *assumption*, since we can then derive the existence of a state σ such that $\sigma \in S$ and $\langle C, \sigma\rangle \rightarrow \sigma'$. Conversely, (2) is useful when our goal is to *prove* $\sigma' \in S_\exists$, since we can prove this goal by simply proving $\sigma \in S$ and $\langle C, \sigma\rangle \rightarrow \sigma'$ for some state σ. Therefore, given a postcondition Q, we derive the postcondition Q' (which we use in our encoding) by translating universally-quantified states (i.e., $\forall\langle\sigma\rangle$) as universal quantifiers with range $S_\forall$ (i.e., $\forall\sigma \in S_\forall$), and existentially-quantified states (i.e., $\exists\langle\sigma\rangle$) as existential quantifiers with range $S_\exists$ (i.e., $\exists\sigma \in S_\exists$).

We follow the same approach for the set of error states (denoted as $S^\bot$), which we (upper- and lower-) bound with $S^\bot_\exists$ and $S^\bot_\forall$. Thus, for the triple $\models [P]\ C\ [Q]$, the Viper encoding generated by our approach has the following shape:

$$\textbf{assume}\ P(S, \varnothing); S_\forall := S; S_\exists := S; S^\bot_\forall := \varnothing; S^\bot_\exists := \varnothing; [\![C]\!]; \textbf{assert}\ Q'(S_\forall, S_\exists, S^\bot_\forall, S^\bot_\exists)$$

where $[\![C]\!]$ constructs $S_\forall$ and $S_\exists$ such that $S_\forall \subseteq sem(C, S) \subseteq S_\exists$ (and similarly for $S^\bot_\forall$ and $S^\bot_\exists$) as we explain below, and Q' is the postcondition obtained from Q as described above.

Soundness. After executing $[\![C]\!]$ in the above encoding, the sets $S_\forall$ and $S_\exists$ (and similarly for $S^\bot_\forall$ and $S^\bot_\exists$) are *underspecified*; $[\![C]\!]$ ensures only that $S_\forall \subseteq sem(C, S) \subseteq S_\exists$. If the generated Viper program is successfully verified, then it is correct for all possible values of $S_\forall$ and $S_\exists$ after $[\![C]\!]$, provided that we started with a set S such that $P(S, \varnothing)$ holds. In particular, it is correct for $S_\forall = sem(C, S) = S_\exists$ (and similarly for $S^\bot_\forall = err(C, S) = S^\bot_\exists$). Thus, successful verification of the generated Viper program implies that, for all S such that $P(S, \varnothing)$ holds, the formula $Q'(sem(C, S), sem(C, S), err(C, S), err(C, S)) = Q(sem(C, S), err(C, S))$ holds, which corresponds exactly to the validity of the hyper-triple $\models [P]\ C\ [Q]$ (Def. 1).

Constructing the lower and upper bounds $S_\forall$ and $S_\exists$. Leveraging properties (1) and (2), our encoding $[\![C]\!]$ tracks two set-valued variables $S_\forall$ and $S_\exists$ that respectively lower and upper bound the set of reachable states. We generate Viper encodings of the following form, where $\langle C, \sigma\rangle \rightarrow \sigma'$ is specialized for each atomic statement C (e.g., assignment, assert-statements, assume-statements) of

[2] In principle, Hypra could also directly generate an SMT encoding. Using an SMT-based tool like Viper provides a more convenient notation and better support for debugging without adding substantial overhead. This makes Viper an ideal candidate for encoding verification conditions.

```
// y := nondet()
assume ∀σ1∈S∀1. ∃σ0, v. σ0∈S∀0 ∧ σ1=σ0[y:=v]
// assume 0 <= y <= 10
assume ∀σ2∈S∀2. σ2∈S∀1 ∧ 0≤σ2(y)≤10
// o := h + y
assume ∀σ3∈S∀3. ∃σ2∈S∀2. σ3=σ2[o:=σ2(h)+σ2(y)]
```

```
// y := nondet()
assume ∀σ0, v. σ0∈S∃0 ⇒ σ0[y:=v]∈S∃1
// assume 0 <= y <= 10
assume ∀σ1∈S∃1. 0≤σ1(y)≤10 ⇒ σ1∈S∃2
// o := h + y
assume ∀σ2∈S∃2. σ2[o:=σ2(h)+σ2(y)]∈S∃3
```

Fig. 5. Viper encodings to compute the lower bound (on the left) and upper bound (on the right) of the set of reachable states, for the body of method `leaky` from Fig. 2. $S^0_\forall$, $S^1_\forall$, $S^2_\forall$, $S^3_\forall$, $S^0_\exists$, $S^1_\exists$, $S^2_\exists$, and $S^3_\exists$ are fresh variables. Additionally, $S^0_\forall$ and $S^0_\exists$ are assumed to satisfy together the precondition of the method (as we explain in Sect. 3.2), and $S^3_\forall$ and $S^3_\exists$ are the sets used to check whether the postcondition holds. $\sigma[x{:=}v]$ denotes the state σ updated with x set to v.

the language, to update the current lower bound $S^n_\forall$ to the next lower bound $S^{n+1}_\forall$, and the current upper bound $S^n_\exists$ to the next upper bound $S^{n+1}_\exists$, for a program statement C:[3]

```
assume ∀σn+1. σn+1∈S∀n+1 ⇒ ∃σn. σn∈S∀n ∧ ⟨C, σn⟩→σn+1 // update lower bound
assume ∀σn, σn+1. σn∈S∃n ∧ ⟨C, σn⟩→σn+1 ⇒ σn+1∈S∃n+1 // update upper bound
```

Starting with $S^n_\forall \subseteq S \subseteq S^n_\exists$, this encoding computes new values for $S^{n+1}_\forall$ and $S^{n+1}_\exists$ such that $S^{n+1}_\forall \subseteq \mathit{sem}(C, S) \subseteq S^{n+1}_\exists$, maintaining the invariant $S^n_\forall \subseteq \mathit{sem}(C, S) \subseteq S^n_\exists$ for all n. Note that the lower bound $S_\forall$ is sufficient to verify $\forall^*$-hyperproperties (safety hyperproperties), while the upper bound $S_\exists$ is sufficient to verify $\exists^*$-hyperproperties. Our tool uses this observation to optimize the encoding when only one kind of reasoning is needed, emitting only the encoding corresponding to $S_\forall$ (for $\forall^*$-hyperproperties) or $S_\exists$ (for $\exists^*$-hyperproperties). Both types of encoding are emitted when verifying both types of hyperproperties or hyperproperties with quantifier alternations.

Fig. 5 shows the concrete encodings generated for the method `leaky` from Fig. 2. The lower-bound encoding, on the left, can be read bottom-up. For any state σ_3 in $S^3_\forall$ at the end of the method, we learn that there exists a state $\sigma_2 \in S^2_\forall$ such that $\sigma_3 = \sigma_2[h := \sigma_2(h) + \sigma_2(y)]$. We then learn that this state σ_2 also belongs to $S^1_\forall$, and satisfies $0 \leq \sigma_2(y) \leq 10$. Finally, we learn that there exists a state $\sigma_0 \in S^0_\forall$ and a value v such that $\sigma_2 = \sigma_0[y := v]$.

In contrast, the upper-bound encoding, on the right, should be read top-down. Starting with any state σ_0 in $S^0_\exists$ and any value (typically provided via a hint), we obtain that $\sigma_0[y := v]$ belongs to $S^1_\exists$. If we can prove that $0 \leq v \leq 10$, we then obtain that σ_1 belongs to $S^2_\exists$. Finally, we obtain that $\sigma_2[o := \sigma_2(h) + \sigma_2(y)]$ belongs to $S^3_\exists$, which can be used as a witness to prove the postcondition of the method.

Encoding conditional statements. For conditional statements, we leverage the fact that

$$\mathit{sem}(\textbf{if } (b) \ \{C_1\} \textbf{ else } \{C_2\}, S) = \mathit{sem}(\textbf{assume } b; C_1, S) \cup \mathit{sem}(\textbf{assume } \neg b; C_2, S)$$

Thus, to construct the lower bound encoding for a conditional statement, we split the current set of program states $S_\forall$ into $S_{\forall_1}$ and $S_{\forall_2}$: $S_{\forall_1}$ is the set of states where b holds, and $S_{\forall_2}$ is the set of states where b does not hold. $S_{\forall_1}$ and $S_{\forall_2}$ are then updated based on the semantics of C_1 and C_2, respectively. We finally compute the union of $S_{\forall_1}$ and $S_{\forall_2}$ to obtain the set of reachable states after the statement. The upper bound encoding for a conditional statement is constructed similarly.

[3]In practice, we have four variables: $S_\forall$ and $S_\exists$, which represent the current bounds, and $S'_\forall$ and $S'_\exists$, which represent the next bounds, and our encoding is of the form `havoc S'; assume ...; S := S'`. We use the superscripts n and $n+1$ to denote the values of these variables at different points in the encoding, to simplify the explanations.

E-matching and the encoding of hints. SMT solvers (such as Z3) used by Viper and other verifiers typically instantiate quantifiers via *E-matching* [Detlefs et al. 2005]. In this approach, every universal quantifier is associated with one or more syntactic matching patterns (also called *triggers*), ground terms that contain the bound variables of the quantifier. The quantifier gets instantiated *only* when the SMT solver's proof search encounters a term that matches its trigger (taking into account equalities). For instance, given the quantifier $\forall x. f(x) > 0$ with trigger $f(x)$, encountering the term $f(5)$ in the proof search will instantiate the quantifier with value 5 for the bound variable x.

Triggers need to be chosen carefully. Overly restrictive triggers may prevent necessary quantifier instantiations, which may cause spurious verification errors. Conversely, overly permissive patterns may, in the worst case, introduce *matching loops*, where each quantifier instantiation produces a term that triggers the next instantiation, causing the SMT solver to diverge.

To avoid matching loops, our encoding uses rather restrictive triggers, as we discuss in more detail in the next subsection. When our chosen triggers are too restrictive, users can initiate additional quantifier instantiations by annotating the input program with hints. These are encoded as applications of a vacuously-true function, which is used as a trigger. Consequently, a hint such as `hint(0,1)` in Fig. 2 causes a quantifier instantiation with the values `0` and `1`. Hints are sometimes needed to instantiate the quantifier in the encoding of non-deterministic assignments; all other quantifiers are instantiated automatically.

Tracking the set of error states. As explained above, on top of tracking (a lower bound and an upper bound of) the set of reachable states, our encoding also tracks a lower bound $S^{\bot}_{\forall}$ and an upper bound $S^{\bot}_{\exists}$ for the set of error states $err(C, S)$ (defined in Def. 1). At the start of every method, the set of error states is initially empty (preconditions, unlike postconditions, are not allowed to quantify over error states). We then grow this set of error states monotonically, because the set of error states for a sequential composition $C_1; C_2$, written $err(C1; C2, S)$, is the union of $err(C_1, S)$ and $err(C_2, sem(C_1, S))$. Similarly, for conditionals, we compute the set of errors for both branches, and take their union. Error states can be generated only by **`assert`** statements: the encoding of **assert** b adds all reachable states that violate b to the set of error states. Finally, the error states arising from loops are handled via loop invariants. For example, to prove that there are no error states after a loop, the invariant must assert the absence of error states after every iteration.

3.2 Combining Lower-Bound and Upper-Bound Encodings

To reason about $\forall^*\exists^*$ and $\exists^*\forall^*$-hyperproperties, we need to combine the two types of encodings, to compute both a lower bound (for universally-quantified states) and an upper bound (for existentially-quantified states) of the set of reachable states. However, naively combining the two encodings can easily lead to matching loops (see Sect. 3.1). In the next two subsubsections, we explain two kinds of potential matching loops and how our encoding avoids them.

3.2.1 Naively Combining the Two Encodings. The most straightforward way to combine the two encodings would be to assume that $S^n_{\forall} = S^n_{\exists}$ at every point of the encoding (i.e., for every n), which is equivalent to tracking a single set of states S^n in each program state rather than a lower and an upper bound. However, this naive combination leads to matching loops, as illustrated by the following encoding for a non-deterministic assignment `y := nonDet()`, where $S^0 = S^0_{\forall} = S^0_{\exists}$ and $S^1 = S^1_{\forall} = S^1_{\exists}$.

```
assume ∀σ₁ ∈ S¹. ∃σ₀, v. σ₀ ∈ S⁰ ∧ σ₁ = σ₀[y := v] // (lower bound)
assume ∀σ₀, v. σ₀ ∈ S⁰ ⇒ σ₀[y := v] ∈ S¹ // (upper bound)
```

At the level of the SMT solver, for any state $\sigma_1 \in S^1$, the lower-bound encoding introduces a new state $\sigma_1[y := v_0] \in S^0$ (for some value v_0). This subsequently triggers the universal quantifiers

```
method simple(x,y:Int) returns (z:Int)
  requires ∀⟨σ1⟩. ∃⟨σ2⟩. σ2(x)>σ1(x)
  requires ∀⟨σ1⟩, ⟨σ2⟩. σ1(y) = σ2(y)
  ensures ∀⟨σ1⟩. ∃⟨σ2⟩. σ2(z)>σ1(z)
{
    z := x + y
}
```

```
...
// next line is required to verify the program
assume S^0_∀ = S^0_∃
assume ∀σ1. σ1∈S^0_∀ ⇒ ∃σ2. σ2∈S^0_∃ ∧ σ2(x)>σ1(x)
assume ∀σ1, σ2. σ1∈S^0_∀ ∧ σ2∈S^0_∀ ⇒ σ1(y) = σ2(y)
⟦ z := x + y ⟧
assert ∀σ1. σ1∈S^1_∀ ⇒ ∃σ2. σ2∈S^1_∃ ∧ σ2(z)>σ1(z)
```

Fig. 6. A simple example that requires both overapproximation and underapproximation reasoning on the left, and its encoding on the right.

in the second **assume** statement, which proves (using $v = v_0$) that $\sigma_1[y := v_0] \in S^1$. This, in turn, triggers the universal quantifier in the first **assume** statement, which introduces a new state $\sigma_1[y := v_0][y := v_1] \in S^0$ (for some value v_1), which is then proven to be in S^1 by the upper-bound encoding, and so on, which results in an infinite cycle of quantifier instantiation.

Preventing this kind of matching loop is the main motivation for our encoding based on upper and lower bounds, that is, we keep $S^n_\forall$ and $S^n_\exists$ as two different sets and do generally *not* equate them (for $n > 0$), which breaks the cycle in the matching loop.

However, equating the lower and upper bound, that is, assuming (via an assume-statement in the Viper program) that $S^0 = S^0_\forall = S^0_\exists$, at the beginning of each method is important for practical examples, as we illustrate with method `simple` in Fig. 6, with its encoding shown on the right. The precondition tells us that for any state σ_1, there exists a state σ_2 such that $\sigma_2(x) > \sigma_1(x)$, and that all states agree on the value of y. Thus, for any state σ_1, there should exist a state σ_2 such that $\sigma_2(z) > \sigma_1(z)$ after the assignment. However, without the assumption that $S^0_\forall = S^0_\exists$ at the beginning of the method, this program would not be verified, as σ_2 belongs to $S^0_\exists$, but not necessarily to $S^0_\forall$, and thus one cannot prove that $\sigma_1(y) = \sigma_2(y)$. We explain next how we can equate the upper and lower bound for S^0 without re-introducing the matching loop illustrated above.

3.2.2 Restricting Quantifier Instantiations for $\forall^*\exists^*$-preconditions. Assuming $S^0_\forall = S^0_\exists$ at the beginning of a method may lead to matching loops if the method precondition contains a $\forall^*\exists^*$-hyperproperty. One example is the first precondition of the method in Fig. 6, which is interpreted as $\forall \sigma_1 \in S^0_\forall. \exists \sigma_2 \in S^0_\exists. \sigma_2(x) > \sigma_1(x)$. The $\forall$-quantifier in the assertion introduces a new state $\sigma_2 \in S^0_\exists$ via the nested $\exists$-quantifier. Since we assume $S^0_\forall = S^0_\exists$, the new state σ_2 can trigger the instantiation of the same $\forall$-quantifier, which in turn can introduce a new state $\sigma_2' \in S^0_\exists$, and so on, leading to a matching loop.

Our solution is to use *limited* and *unlimited* functions [Leino and Monahan 2009] to control quantifier instantiations, to allow as many existentially-quantified states as possible to instantiate universal quantifiers, while avoiding matching loops. Concretely, when our tool translates a hyper-assertion with a universal state-quantifier such as $\forall\langle\sigma\rangle. P$ into Viper, it checks whether P contains an existential state-quantifier: If so, the $\forall$-quantifier is encoded with the *most restrictive* trigger, so that it can be instantiated only with those existentially-quantified states that do not occur under a universal quantifier. Otherwise, the $\forall$-quantifier is encoded with a more permissive trigger, allowing it to be instantiated by all existentially-quantified states.

The effect of our solution is represented visually on Fig. 7. Each node ($\exists^+$, $\exists^+\forall^+$, $\forall^+\exists^+$, $\forall^+$) represents the shape of a possible part of the precondition (the figure omits shapes with more than two quantifiers for simplicity). The blue arrows show the instantiations *enabled* by our tool, while the red dashed arrow shows the instantiation *prevented* by our tool via a restrictive trigger, since

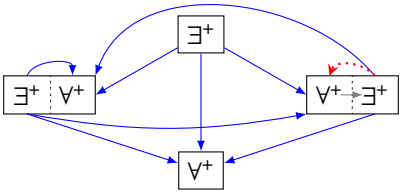

Fig. 7. Representation of the quantifier instantiations allowed by the chosen triggers. Each node represents the shape of a possible part of the precondition; for simplicity, we omit shapes with more than two quantifiers here. The blue arrows show the existentially-quantified states that can be used to instantiate universal quantifiers, and the red arrow shows the instantiations our chosen triggers prevent. The grey arrow shows how instantiating the $\forall^+$-quantifiers in a $\forall^+\exists^+$-hyperproperty produces existentially-quantified states, which can subsequently be used to instantiate universal quantifiers. The acyclicity of the graph ensures the absence of matching loops among the involved quantifiers.

those instantiations can lead to matching loops. For example, states introduced by $\exists^+$-quantifiers can be used to instantiate the $\forall^+$-quantifiers of a $\forall^+\exists^+$-hyperproperty, which can in turn be used to instantiate the $\forall^+$-quantifiers of a $\forall^+$-hyperproperty. As a more concrete example, to verify the method `simple` from Fig. 6, the state σ_2 coming from the existential quantifier of the first precondition $\forall\langle\sigma_1\rangle.\, \exists\langle\sigma_2\rangle.\, \sigma_2(x) > \sigma_1(x)$ can be used to instantiate a $\forall$-quantifier of the second precondition $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(y) = \sigma_2(y)$, since there is a blue arrow from the right of the node $\forall^+\exists^+$ to the node $\forall^+$.

Crucially, states introduced by existential quantifiers that are nested under universal quantifiers *cannot* be used to instantiate the universal quantifiers in $\forall^+\exists^+$-hyperproperties (as represented by the red dotted arrow), since this would create a cycle and thus lead to a matching loop, as illustrated above with the first precondition of the method `simple`. As can be seen in Fig. 7, preventing this instantiation makes the graph acyclic, which ensures the absence of matching loops.

4 Verification Conditions for Loops

In the previous section, we described the verification conditions generated by our verifier for loop-free statements. In this section, we describe how to generate verification conditions for while loops. We first describe, in Sect. 4.1, the different rules offered by Hyper Hoare Logic to reason about while loops, how we can derive verification conditions from them, and how our verifier automatically selects the right rule(s) to apply, based on the context. In Sect. 4.2, we discuss one such particular rule, the *While*-$\forall^*\exists^*$ rule, and show that it cannot be used directly for our purpose; a *naive* encoding based on this rule would be unsound. We then present and prove sound (in Isabelle/HOL [Nipkow et al. 2002]) a novel loop rule, suitable for automated deductive verification, which can be used in the same context. Finally, in Sect. 4.3, we present a technique to automatically frame information around the loop, which overcomes a limitation of these loop rules, and leads to more succinct loop invariants.

4.1 Automatically Generating Verification Conditions

Reasoning about loops in a relational setting is notoriously hard. In the context of deductive verification, our goal is to automatically reason about while loops, while keeping the amount of proof annotations needed from the user to a minimum. As illustrated in Fig. 4, this means that the user should only provide a loop invariant, and optionally a loop variant (**`decreases`** clause).

$$\frac{I \models \mathit{low}(b) \quad \models [I \wedge \Box b]\ C\ [I]}{\models [I]\ \textbf{while}\ (b)\ \{C\}\ [(I \vee \Box\bot) \wedge \Box(\neg b)]}\ (\mathit{WhileSync})$$

$$\frac{I \models \mathit{low}(b) \quad \models_{\Downarrow} [I \wedge \Box(b \wedge e = t)]\ C\ [I \wedge \Box(e \prec t)] \quad \prec \text{ well-founded} \quad t \notin \mathit{fv}(I) \cup \mathit{mod}(C)}{\models_{\Downarrow} [I]\ \textbf{while}\ (b)\ \{C\}\ [I \wedge \Box(\neg b)]}\ (\mathit{WhileSyncTerm})$$

$$\frac{\models [I]\ \textbf{if}\ (b)\ \{C\}\ [I] \quad \models [I]\ \textbf{assume}\ \neg b\ [Q] \quad \text{no } \forall\langle\_\rangle \text{ after any } \exists \text{ in } Q}{\models [I]\ \textbf{while}\ (b)\ \{C\}\ [Q]}\ (\mathit{While}\text{-}\forall^*\exists^*)$$

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

Fig. 8. Rules from Hyper Hoare Logic [Dardinier and Müller 2024] to reason about while loops. In the rules *WhileSyncTerm* and *While*-∃, the order $\prec$ must be *well-founded* (wf). Moreover, $\mathit{low}(b) \triangleq (\forall\langle\sigma\rangle, \langle\sigma'\rangle.\, b(\sigma) = b(\sigma'))$ and $\Box(b) \triangleq (\forall\langle\sigma\rangle.\, b(\sigma))$. Finally, $\models_{\Downarrow} [P]\ C\ [Q]$ corresponds to a *terminating* hyper-triple. Terminating hyper-triples are stronger than normal hyper-triples, in that they additionally require the existence of a terminating execution from any initial state.

(a)	(b)	(c)	(d)
assert $I(S, \emptyset)$	**assert** $I(S, \emptyset)$	**assert** $I(S, \emptyset)$	**assert** $\exists\sigma.\, P_\sigma(S, \emptyset)$
S_p := S	S_p := S	S_p := S	S_p := S
$S_0^\bot$:= $S^\bot$	$S_0^\bot$:= $S^\bot$	$S_0^\bot$:= $S^\bot$	$S_0^\bot$:= $S^\bot$
havoc $S, S^\bot$	**havoc** $S, S^\bot$	**havoc** $S, S^\bot$	**havoc** $S, S^\bot$
assume $I(S, S^\bot)$	**assume** $I(S, S^\bot)$	**assume** $I(S, S^\bot)$	**assume** $\exists\sigma.\, P_\sigma(S, S^\bot) \wedge b(\sigma) \wedge v = e(\sigma)$
assert $\mathit{low}(b)$	**assert** $\mathit{low}(b)$	⟦**if** (b) {C}⟧	⟦**if** (b) {C}⟧
assume $\Box b$	**assume** $\Box(b \wedge e = t)$		**assert** $\exists\sigma.\, P_\sigma(S, S^\bot) \wedge 0 \le e(\sigma) \prec v$
⟦C⟧	⟦C⟧		**havoc** $S, S^\bot$
			var σ_0: State
	assert $\Box(0 \le e \prec t)$		**assume** $P_{\sigma_0}(S, S^\bot)$
			⟦**while** (b) {C}⟧
assert $I(S, S^\bot)$	**assert** $I(S, S^\bot)$	**assert** $I(S, S^\bot)$	**assert** $P_{\sigma_0}(S, S^\bot)$
havoc $S, S^\bot$	**havoc** $S, S^\bot$	**havoc** $S, S^\bot$	**havoc** $S, S^\bot$
assume $F(S_p, S)$	**assume** $F(S_p, S)$	**assume** $F(S_p, S)$	**assume** $F(S_p, S)$
assume $I(S, S^\bot) \vee \Box\bot$	**assume** $I(S, S^\bot)$	**assume** $\Theta_{\neg b}(I)$	**assume** $\exists\sigma.\, P_\sigma(S, S^\bot)$
$S^\bot$:= $S_0^\bot \bigcup S^\bot$	$S^\bot$:= $S_0^\bot \bigcup S^\bot$	$S^\bot$:= $S_0^\bot \bigcup S^\bot$	$S^\bot$:= $S_0^\bot \bigcup S^\bot$
assume $\Box(\neg b)$	**assume** $\Box(\neg b)$	⟦**assume** $\neg b$⟧	**assume** $\Box(\neg b)$

Fig. 9. Viper encodings of loops based on (a) the *WhileSync* rule, (b) the *WhileSyncTerm* rule, (c) the novel rule for $\forall^*\exists^*$-hyperproperties and (d) the weakened *While*-∃ rule. To avoid clutter, we use S to represent both $S_\forall$ and $S_\exists$, and $S^\bot$ to represent both $S_\forall^\bot$ and $S_\exists^\bot$. We also use the notation ⟦C⟧ to refer to the encoding of the command C. In the loop encodings, b is the loop guard, C is the loop body, v and t are fresh variables, S_p is an auxiliary variable recording the value of the set of states before the loop (see Sect. 4.3), e is the loop variant, I and P_σ are loop invariants encoded as a predicate dependent on S and $S^\bot$. Moreover, $\mathit{low}(b) \triangleq (\forall\langle\sigma\rangle, \langle\sigma'\rangle.\, b(\sigma) = b(\sigma'))$, $\Box(b) \triangleq (\forall\langle\sigma\rangle.\, b(\sigma))$, and $F(S_p, S)$ corresponds to automatic framing as described in Sect. 4.3.

Fig. 8 shows the four main rules offered by Hyper Hoare Logic to reason about while loops, where $\mathit{low}(b)$ means that the expression b has the same value in all states (formally $\mathit{low}(b) \triangleq (\forall\langle\sigma\rangle, \langle\sigma'\rangle.\, b(\sigma) = b(\sigma'))$), $\Box b$ means that the expression b holds in all states (formally $\Box b \triangleq (\forall\langle\sigma\rangle.\, b(\sigma))$), and $\models_{\Downarrow} [P]\ C\ [Q]$ corresponds to a *terminating* hyper-triple. Terminating hyper-triples are stronger than normal hyper-triples, in that they additionally require the existence of a

terminating execution from any initial state. In our tool, we ensure this requirement by proving that all loops in C terminate, through the use of well-founded loop variants.[4] As shown by Fig. 8, all four rules use a loop invariant: I in the first three loop rules, and $\exists\langle\sigma\rangle.\, P_\sigma$ in the last rule. Moreover, those rules are non-obvious, which makes it hard for users to know which rule to apply in which context.

In the following, we explain the role of the different rules, how we derive verification conditions from them, and how our verifier automatically chooses the relevant rule(s), based on the user-provided loop invariant and optional loop variant. The Viper encodings of loops based on these rules are shown in Fig. 9.

Synchronized loop rules. The two first rules, *WhileSync* and *WhileSyncTerm*, apply when all executions exit the loop simultaneously. The key difference between the two rules can be seen in the postconditions of their conclusions: On top of the fact that all states satisfy the negation of the loop guard ($\Box(\neg b)$), the rule *WhileSyncTerm* allows us to assume that the relational invariant I holds after the loop, whereas the rule *WhileSync* allows us to assume only that $I \vee \Box\bot$ holds after the loop, which is weaker than I. The $\Box\bot$ disjunct, which corresponds to the case where the loop does not terminate, is problematic when we want to prove postconditions with top-level existentially-quantified states. In this case, we need to use the rule *WhileSyncTerm*, which requires us to prove that the loop terminates. The latter can be achieved by proving that a well-founded variant e strictly decreases after every iteration.

Verification conditions can be easily derived from these two rules. First, for both rules, we check that the user-provided loop invariant I entails $low(b)$. Then, for the rule *WhileSync*, we separately check the triple $\models\ [I \wedge \Box b]\ C\ [I \vee \Box\bot]$, as described in Sect. 3. For the rule *WhileSyncTerm*, we instantiate e with the user-provided loop variant (required to be an integer expression), and separately check the triple $\models_{\Downarrow}\ [I \wedge \Box(b \wedge e = t)]\ C\ [I \wedge \Box(0 \leq e < t)]$, where t is a fresh variable. The check $0 \leq e$ ensures that the user-provided variant is well-founded. To ensure that this triple is a terminating hyper-triple, we check (syntactically) that all loops within C are annotated with **`decreases`** clauses, which ensures termination (provided that verification is successful). Finally, for both rules, we assert that the loop invariant I holds before the loop, and assume that $(I \vee \Box\bot) \wedge \Box(\neg b)$ (rule *WhileSync*) or $I \wedge \Box(\neg b)$ (rule *WhileSyncTerm*) holds after the loop. The Viper encodings of loops based on these two rules can be found in Fig. 9a and Fig. 9b, respectively.

Non-synchronized loop rules. The two remaining loop rules from Fig. 8, *While*-$\forall^*\exists^*$ and *While*-$\exists$, can be applied when different executions might exit the loop at different times. In this case, our premises are more complex: We need to reason about the *unrollings* of the while loop, which we achieve by proving a loop invariant over **if** (b) $\{C\}$ (in contrast to C for the synchronized rules). Deriving verification conditions from the rule *While*-$\forall^*\exists^*$ is non-trivial, as we explain in Sect. 4.2. For the rule *While*-$\exists$, assuming that the user-provided loop invariant I is of the form $\exists\langle\sigma\rangle.\, P_\sigma$ (we write P_σ to emphasize that this hyper-assertion can mention σ),[5] and that the user provided a loop

[4] In theory, we also need to check that **`assume`** statements do not break this property. By default, our tool leaves this responsibility to the user, since **`assume`** statements are typically used to restrict non-deterministic assignments to the right range (as done in the example from Fig. 4), which does not break this property. However, our tool provides the more conservative option, disabled by default, to check the absence of **`assume`** statements within statements whose termination is required.

[5] If I is not of the shape $\exists\langle\sigma\rangle.\, P_\sigma$, and no other rule is applicable, our tool emits an error message to inform the user that the program cannot be verified.

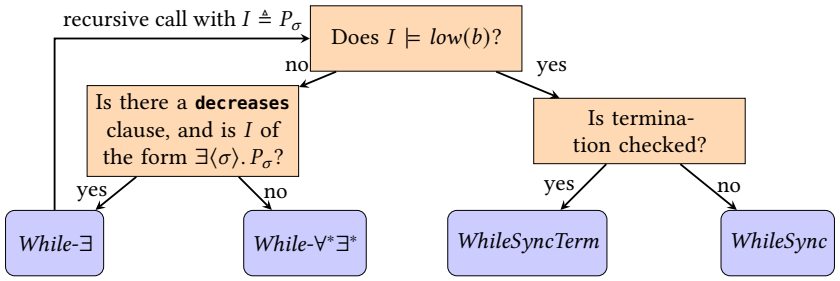

Fig. 10. Automatic loop rule selection to verify a loop **while** (b) $\{C\}$ with the invariant I. The first choice checks whether all executions perform the same number of loop iterations. The next choice is based on a subtle aspect of the loop's termination: The branch on the left checks whether the given loop performs a finite number of iterations, but does *not* require nested loops to terminate; this is sufficient to apply *While-∃*. In contrast, the branch on the right requires the loop and all nested loops to terminate, as required by the rule *WhileSyncTerm*. While the top-level choice is made semantically by the encoding, the next-level choices are determined syntactically based on the presence of **decreases** clauses. The edge from *While-∃* back to the first choice reflects the second premise of this rule, $\forall\sigma. \models [P_\sigma]$ **while** (b) $\{C\}$ $[P_\sigma]$: To check that this premise holds, our tool recursively calls the rule selection procedure, which will automatically select a new loop rule adapted to the new loop invariant P_σ.

variant e, we apply the following weakened version of the rule:

$$\frac{\forall v. \models [\exists\langle\sigma\rangle. P_\sigma \wedge b(\sigma) \wedge v = e(\sigma)]\ \textbf{if}\ (b)\ \{C\}\ [\exists\langle\sigma\rangle. P_\sigma \wedge 0 \leq e(\sigma) < v] \quad \forall\sigma. \models [P_\sigma]\ \textbf{while}\ (b)\ \{C\}\ [P_\sigma]}{\models \underbrace{[\exists\langle\sigma\rangle. P_\sigma]}_{I}\ \textbf{while}\ (b)\ \{C\}\ [\underbrace{(\exists\langle\sigma\rangle. P_\sigma)}_{I} \wedge \square(\neg b)]}$$

As before, this version specializes the well-founded order $\prec$ to be the canonical well-founded order over natural numbers. Note that, in both premises, v and σ are *meta-variables*, i.e., there is not one value of v (in the first premise) or σ (in the second premise) per state, but rather there is one per *set* of states.

In practice, to check the first premise, we use a fresh unconstrained variable v, and assume that the set of states and the variable v together satisfy the precondition $\exists\langle\sigma\rangle. P_\sigma \wedge b(\sigma) \wedge v = e(\sigma)$, and check (after the encoding of **if** (b) $\{C\}$) that the set of states and the variable v together satisfy the postcondition $\exists\langle\sigma\rangle. P_\sigma \wedge 0 \leq e(\sigma) \prec v$. Checking the second premise is more complicated, since it requires reasoning about the same while loop. However, note that the precondition (and postcondition) of this premise, P_σ, is smaller than the precondition (and postcondition) of the conclusion of the rule, $\exists\langle\sigma\rangle. P_\sigma$. Our tool automatically generates the verification conditions for this premise, using the approach described in this section, by automatically selecting the right loop rule based on the new loop invariant P_σ and the same loop variant e. The Viper encoding of loops based on this rule can be found in Fig. 9d.

Automatically selecting the right loop rule(s). As explained at the start of this section, using only the user-provided loop invariant I and optional loop variant e, our tool automatically selects the right loop rule(s) to apply, as depicted in Fig. 10. First, we check (semantically in our encoding) whether the loop invariant guarantees that all executions will exit the loop simultaneously, that is, whether $I \models low(b)$ holds. If so, we apply one of the two synchronized loop rules, *WhileSync* or *WhileSyncTerm*, depending on whether the user provided a loop variant for this loop and all loops nested within. Compared to non-synchronized loop rules, these two rules, when applicable, have

weaker premises (i.e., their premises can be derived from the premises of the non-synchronized loop rules) and stronger conclusions (e.g., their postconditions after the while loop are at least as strong as those of the non-synchronized loop rules). Therefore, the synchronized loop rules are always better than the non-synchronized loop rules when they are applicable. The rule *WhileSyncTerm* is the most powerful (when it applies), because its premise only requires reasoning about C, which is easier than reasoning about **if** (b) $\{C\}$, and the postcondition of its conclusion, $I \wedge \Box(\neg b)$, is not weaker than the postcondition given by any other rule (Sect. 4.2 will make clearer why the postcondition of the rule *While*-$\forall^*\exists^*$ is weaker). When termination is not checked (i.e., no loop variant is provided), the choice is only between the rules *WhileSync* and *While*-$\forall^*\exists^*$, and our tool applies *WhileSync* whenever possible, for similar reasons.

When $I \not\models \mathit{low}(b)$, we apply one of the two non-synchronized loop rules, *While*-$\forall^*\exists^*$ or *While*-$\exists$, depending on the shape of the loop invariant I. When I is of the form $\forall^+\exists^*$, we can apply only the rule *While*-$\forall^*\exists^*$, because our invariant is not of the form $\exists\langle\sigma\rangle.\, P$, required by the rule *While*-$\exists$. Similarly, when I is of the form $\exists^+\forall^+$, we can apply only the rule *While*-$\exists$, because I does not satisfy the syntactic restriction from the rule *While*-$\forall^*\exists^*$. Thus, there exists a *choice* between those two loop rules only when I has the shape $\exists^+$ and a loop variant is provided (otherwise the rule *While*-$\exists$ cannot be applied). In this case, the rule *While*-$\exists$ is more powerful, since it easily allows proving that the existentially-quantified states in I will still exist after the while loop, thanks to the loop variant. As a concrete example, let $I \triangleq \exists\langle\sigma\rangle.\, \sigma(x) = \sigma(y)$. We can use the rule *While*-$\exists$ with $P_\sigma \triangleq (\sigma(x) = \sigma(y))$, which gives us the desired postcondition $\exists\langle\sigma\rangle.\, \sigma(x) = \sigma(y)$ in its conclusion. In contrast, the postcondition of the conclusion of the rule *While*-$\forall^*\exists^*$ is some hyper-assertion Q, such that $\models [\exists\langle\sigma\rangle.\, \sigma(x) = \sigma(y)]$ **assume** $\neg b$ $[Q]$ holds. In particular, we can get our desired postcondition $\exists\langle\sigma\rangle.\, \sigma(x) = \sigma(y)$ only if $\sigma(x) = \sigma(y)$ implies $\neg b$, which will typically not be the case (because $\exists\langle\sigma\rangle.\, \sigma(x) = \sigma(y)$ is our loop invariant, which has to already hold *before* the loop). Thus, when applicable, our tool applies the rule *While*-$\exists$ over the rule *While*-$\forall^*\exists^*$, which then recursively applies the same automatic loop rule selection with the smaller loop invariant P, as shown in Fig. 10.

4.2 $\forall^*\exists^*$-Hyperproperties

In some cases, the only loop rule that can be applied is the rule *While*-$\forall^*\exists^*$. Automating this rule is surprisingly not straightforward. Checking the premise $\models [I]$ **if** (b) $\{C\}$ $[I]$ of the rule is easy, as it can be checked separately using the user-provided loop invariant I. However, deriving from the loop invariant I a suitable postcondition Q that satisfies the syntactic restriction is more challenging. In the following, we first show why the naive *semantic* approach for deriving Q does not work, and then discuss our solution, which derives Q *syntactically* from I.

Naively deriving Q semantically is unsound. A natural idea is to check the syntactic restriction (no universal state-quantifier should occur under an existential quantifier) on I instead of Q, and then to derive Q from I *semantically*, i.e., we can obtain the postcondition Q by considering a fresh set of states after the loop, assuming that it satisfies I, and then encoding **assume** $\neg b$. We cannot check the syntactic restriction on Q (as mandated by the rule) directly, since Q is not a *syntactic* hyper-assertion. Unfortunately, this natural idea surprisingly results in an unsound encoding, as we illustrate next.

As an example unsoundly accepted by this naive encoding, consider the method `naive_encoding` from Fig. 11. Depending on the value of t, this method will either loop forever (if $t = 1$) or simply increment x until $x = n$ (if $n \geq 0$ and $t = 2$). Our precondition[6] requires the existence of a state that

[6]Note that no precondition is actually required for the naive encoding to be unsound on this example, but we use one to simplify the explanations.

```
method naive_encoding(t: Int, n: Int) returns (x: Int)
    requires ∃⟨σ1⟩. σ1(t) = 1
    requires ∀v. v ≥ 0 ⇒ ∃⟨σ2⟩. σ2(t) = 2 ∧ σ2(n) = v
    ensures ∃v. ∀⟨σ⟩. σ(x) ≤ v // this postcondition does not hold
{
    x := 0
    while (t = 1 || x < n)
        invariant ∃⟨σ1⟩. σ1(t) = 1
        invariant ∀⟨σ1⟩, ⟨σ2⟩. σ1(t) = 1 ⇒ σ1(x) ≥ σ2(x)
    {
        x := x + 1
    }
}
```

Fig. 11. An example showing why a *naive* encoding based on the rule *While*-$\forall^*\exists^*$ would be unsound.

will loop forever ($t = 1$), and, for all possible non-negative values v of n, the existence of a state that will do n iterations until $x = n$. The postcondition, which does not hold, requires the existence of an upper bound v for the value of x in all states.

To understand why this postcondition does not hold, consider a set of states S that satisfies the precondition, i.e., S contains at least one state with $t = 1$, and, for each natural number v, S contains a state σ where $\sigma(n) = v$ and $\sigma(t) = 2$. Moreover, let S' be the set of states after the loop. For all states with $t = 2$, x will be equal to n after the while loop, i.e., $\forall \sigma' \in S'. \sigma'(t) = 2 \Rightarrow \sigma'(x) = \sigma'(n)$. Thus, for each natural number v, S' contains a state σ' where $\sigma'(x) = \sigma'(n) = v$ (and $\sigma'(t) = 2$). In particular, this implies that the set of values $\{\sigma'(x) \mid \sigma' \in S'\}$ does not have an upper bound. This contradicts the postcondition, which expresses the existence of such an upper bound.

We now explain why the naive encoding described above accepts this program. The first premise of the rule *While*-$\forall^*\exists^*$, $\models [I]$ **if** $(t = 1 \vee x < n)$ $\{x := x+1\}$ $[I]$, holds, since any state σ_1 with $t = 1$ will enter the if-branch, and thus σ_1 will keep having the maximal value (among all executions) for x. Moreover, the loop invariant I satisfies the syntactic restriction (no $\forall\langle\_\rangle$ appears under any existential quantifier), and I clearly holds before the loop, since $x = 0$ in all states. Finally, let us consider what happens after the loop, and why this encoding allows us to derive the wrong postcondition. Let S be a set of states that satisfies the loop invariant I, and let S' be the set of states obtained by executing **assume** $\neg(t = 1 \vee x < n)$ in all states from S. Note that S' corresponds to the subset of states from S that satisfy $t \neq 1 \wedge x \geq n$. From I, we learn that there is a state σ_1 in S where $t = 1$, and that this state σ_1 has the maximum value for x among all states in S. Thus, there exists an upper bound v for the value of x in all states from S, namely $v \triangleq \sigma_1(x)$. Since S' is a subset of S, this upper bound v is also an upper bound for the value of x in all states from S', which corresponds to the incorrect postcondition.

A new rule for automating $\forall^*\exists^*$-hyperproperties. The previous example shows that deriving the postcondition Q *semantically* from I, while checking the *syntactic* restriction on the loop invariant I, is unsound. We solve this issue by deriving the postcondition Q *syntactically* from I while enforcing the *syntactic* restriction on I. This allows us to obtain a sound rule, which can be automated in a straightforward way as shown by Fig. 9c. We obtain the postcondition from I, which we write $\Theta_{\neg b}(I)$, by recursively replacing all instances of $\exists\langle\sigma\rangle. P$ with $\exists\sigma. P \wedge (\neg b \Rightarrow \langle\sigma\rangle)$.[7] That is, the postcondition ensures that the existentially-quantified states in I *exist*, but they are not guaranteed

[7]We overload the notation $\langle\sigma\rangle$ to mean $\lambda S. \sigma \in S$, i.e., the formula is equivalent to $\lambda S. \exists\sigma. P \wedge (\neg b \Rightarrow \sigma \in S)$.

to belong to the set of states after the loop: they belong to the set of states after the loop if they satisfy the negation $\neg b$ of the loop guard. Although $\Theta_{\neg b}(I)$ is not a well-formed hyper-assertion according to the syntax in Sect. 2.2, this is not an issue since $\Theta_{\neg b}(I)$ is not an annotation in a user-written program but only appears in the generated Viper program. In the example from Fig. 11, we obtain from I the postcondition $\Theta_{\neg b}(I) = (\exists \sigma_1.\, \sigma_1(t) = 1 \wedge (\neg(\sigma_1(t) = 1 \vee \sigma_1(x) < \sigma_1(n)) \Rightarrow \langle \sigma_1 \rangle)) \wedge (\forall \langle \sigma_1 \rangle, \langle \sigma_2 \rangle.\, \sigma_1(t) = 1 \Rightarrow \sigma_1(x) \geq \sigma_2(x))$, which does not entail the wrong postcondition anymore. Indeed, while we still learn the existence of a state σ_1 where $t = 1$, we do not learn that σ_1 belongs to the set of states after the loop, because we cannot prove $\neg(\sigma_1(t) = 1 \vee \sigma_1(x) < \sigma_1(n))$, and, thus, we cannot conclude that $\forall \langle \sigma_2 \rangle.\, \sigma_1(x) \geq \sigma_2(x)$.

We have proven in Isabelle/HOL [Nipkow et al. 2002] that this novel rule, which our tool leverages, is sound:

Theorem 1. ***Novel loop rule for $\forall^*\exists^*$-hyperproperties.*** *Let C be a program statement, b a program expression, and I a (syntactic) hyper-assertion such that I contains no $\forall\langle\_\rangle$ after any $\exists$. If $\models [I]$ **if** (b) $\{C\}$ $[I]$ holds, then $\models [I]$ **while** (b) $\{C\}$ $[\Theta_{\neg b}(I) \wedge \square(\neg b)]$ holds.*

Proof sketch. To prove this result, we use the fact that $sem(\textbf{while}\ (b)\ \{C\}, S)$, the semantics of the while loop given a set of initial states S, can be seen as the limit of $sem([\textbf{if}\ (b)\ \{C\}]^n; \textbf{assume}\ \neg b, S)$ as n goes to infinity, where $[\textbf{if}\ (b)\ \{C\}^n]$ represents the statement $\textbf{if}\ (b)\ \{C\}$ sequentially composed with itself n times. More formally:

$$sem(\textbf{while}\ (b)\ \{C\}, S) = \bigcup_{n \in \mathbb{N}} sem([\textbf{if}\ (b)\ \{C\}]^n; \textbf{assume}\ \neg b, S) \qquad (*)$$

In other words, every state after the loop must have exited the loop after n iterations, for some n. In particular, note that the sequence of sets $(sem([\textbf{if}\ (b)\ \{C\}]^n; \textbf{assume}\ \neg b, S))_{n \in \mathbb{N}}$ is non-decreasing. That is, the set of states that exit the loop in the first n iterations can only grow when n grows.

We then prove the following property $\mathcal{P}(I)$, by structural induction over the syntactic hyper-assertion I: "For any non-decreasing sequence $(S_n)_{n \in \mathbb{N}}$ of set of states, if I contains no $\forall\langle\_\rangle$ after any $\exists$, and if $\forall n.\, S_n \models \Theta_{\neg b}(I)$, then $(\bigcup_n S_n) \models \Theta_{\neg b}(I)$ holds". The theorem follows from this property and the aforementioned identity (*). In the following, we discuss four cases of the induction; all other cases are trivial.

The cases for $\mathcal{P}(\exists y.\, I)$ and $\mathcal{P}(\exists \langle \sigma \rangle.\, I)$ are straightforward: By the syntactic restriction, we know that I contains no $\forall\langle\_\rangle$, and so does $\Theta_{\neg b}(I)$. Intuitively, this means that $\Theta_{\neg b}(I)$ only cares about the *existence* of states, and thus $\Theta_{\neg b}(I)$ grows monotonically (which can be proven by an additional trivial induction on I): If it is satisfied by a set of states, then it will be satisfied by any superset of this set. Since it is satisfied by all S_n, it is also satisfied by their union.

For the case $\mathcal{P}(\forall \langle \sigma \rangle.\, I)$, we get to assume (1) $\mathcal{P}(I)$ and (2) $\forall n.\, S_n \models \forall \langle \sigma \rangle.\, \Theta_{\neg b}(I)$, and we want to prove $(\bigcup_n S_n) \models \forall \langle \sigma \rangle.\, \Theta_{\neg b}(I)$. To prove this, let σ be a state in $\bigcup_n S_n$, and let us prove that $(\bigcup_n S_n), \sigma \models \Theta_{\neg b}(I)$ (which informally means that the previously existentially-quantified state σ is instantiated in $\Theta_{\neg b}(I)$ to the concrete state)[8]. Because $\sigma \in \bigcup_n S_n$, there exists a k such that $\sigma \in S_k$. Let S' such that $\forall n.\, S'_n = S_{n+k}$. Because $\forall n.\, S'_n, \sigma \models \Theta_{\neg b}(I)$, we can use the induction hypothesis $\mathcal{P}(I)$ to get that $(\bigcup_n S'_n) \models \Theta_{\neg b}(I)$. Finally, notice that $(\bigcup_n S_n) = (\bigcup_n S'_n)$, because S is non-decreasing, which concludes the case.

For the case $\mathcal{P}(I_1 \vee I_2)$, we get to assume (1) $\mathcal{P}(I_1)$, (2) $\mathcal{P}(I_2)$, and (3) $\forall n.\, S_n \models \Theta_{\neg b}(I_1) \vee \Theta_{\neg b}(I_2)$, and we want to prove $(\bigcup_n S_n) \models \Theta_{\neg b}(I_1) \vee \Theta_{\neg b}(I_2)$. By (3), we know that $\Theta_{\neg b}(I_1) \vee \Theta_{\neg b}(I_2)$ is true infinitely often, which implies that either $\Theta_{\neg b}(I_1)$ is true infinitely often, or $\Theta_{\neg b}(I_2)$ is true infinitely often. Without loss of generality, let us assume that $\Theta_{\neg b}(I_1)$ is true infinitely often, and let S' be an infinite subsequence of S such that $\forall n.\, S'_n \models \Theta_{\neg b}(I_1)$. By the induction hypothesis $\mathcal{P}(I_1)$, we get

[8] In our mechanization, we use de Bruijn indices to handle quantifiers, which we ignore in this paper for simplicity.

```
method framing1(x: Int) returns (y: Int)
    requires ∀⟨σ1⟩, ⟨σ2⟩. σ1(x) = σ2(x)
    requires ∀⟨σ⟩. σ(x) ≥ 0
    ensures ∀⟨σ1⟩, ⟨σ2⟩. σ1(y) = σ2(y)
{
    y := 0
    while (y < x)
        invariant ∀⟨σ⟩. σ(y) ≤ σ(x)

    { y := y + 1 }
}
```

```
method framing2(x: Int) returns (y: Int)
    requires ∃⟨σ⟩. ∀⟨σ'⟩. σ(x) ≥ σ'(x)
    requires ∀⟨σ⟩. σ(x) ≥ 0
    ensures ∃⟨σ⟩. ∀⟨σ'⟩. σ(y) ≥ σ'(y)
{
    y := 0
    while (y < x)
        invariant ∀⟨σ⟩. σ(y) ≤ σ(x)
        decreases x - y
    { y := y + 1 }
}
```

Fig. 12. An example from Hypra that requires automatic framing to be successfully verified.

that $(\bigcup_n S'_n) \models \Theta_{\neg b}(I_1)$. Moreover, because S is non-decreasing, we get that $(\bigcup_n S_n) = (\bigcup_n S'_n)$, which concludes the case. □

4.3 Automatic Framing

In the previous subsections, we have shown how we derived verification conditions from the loop rules offered by Hyper Hoare Logic. However, using those loop rules on their own (and not in conjunction with other rules as we show below) has the limitation that only the information provided by the loop invariant is preserved, as we illustrate with the examples in Fig. 12. Consider the method `framing1` on the left of the figure, which increments y in a loop until $x = y$. We want to prove that if x has the same initial value in all executions, then y will have the same final value in all executions. Using the standard (unary) loop invariant $\forall\langle\sigma\rangle.\, \sigma(y) \leq \sigma(x)$, we can easily prove that, after the loop, $x = y$ in all states (1). Moreover, since all executions have the same value for x before the loop, and since x is not modified by the loop, all executions will still have the same value for x after the loop (2). By conjoining (1) and (2), we get the postcondition.

Unfortunately, the loop encodings presented so far are only able to prove (1), but not (2), since they assume only (a property derived from) the loop invariant after the loop. Because our loop invariant does not mention that x has the same value in all executions, this piece of information is lost after the loop.

One way to solve this particular problem is to add this information to the loop invariant, by conjoining $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(x) = \sigma_2(x)$ to it. This solution is cumbersome for the user, who is required to write longer invariants, by adding information not relevant for the loop (but only for the postcondition later). Another way to solve this issue is to use the following rule from Hyper Hoare Logic (where $mod(C)$ represents the variables modified by C and $fv(F)$ the (program) variables that appear in F), which allows propagating information about variables not modified by the loop after the loop:

$$\frac{\models [P]\ C\ [Q] \quad \text{no } \exists\langle\_\rangle \text{ in } F \quad mod(C) \cap fv(F) = \varnothing}{\models [P \wedge F]\ C\ [Q \wedge F]}\ (\mathit{FrameSafe})$$

Since x is not modified by the loop, we can use this rule with $F \triangleq (\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(x) = \sigma_2(x))$ to solve our issue. Our goal is to use this rule to *automatically* frame information around the loop, without requiring the user to provide F.

We achieve this by recording the set of states before the loop in an auxiliary variable S_p, and by adding (after the loop) the assumption that, for each state in the set of states after the loop, there must exist a state in S_p with the same values for all variables not modified by C:

```
S_p := S
... // loop encoding
assume ∀σ' ∈ S. ∃σ ∈ S_p. (∀x. x ∉ fv(C) ⇒ σ(x) = σ'(x)) // (F-OX)
```

Intuitively, adding this assumption is sound because every state σ' after the loop corresponds to the final state of an execution of C in an initial state σ from S_p, and thus σ and σ' must have the same values for the variables not modified by C. Formally, this encoding is justified by the following straightforward lemma:

Lemma 1. $\forall \sigma' \in \mathit{sem}(C, S). \exists \sigma \in S. \forall x \notin \mathit{mod}(C). \sigma(x) = \sigma'(x)$

This encoding is stronger than *any* possible application of the rule *FrameSafe*, since the former logically implies the latter (for any frame F). To see why it solves the issue from our example, consider two states σ'_1 and σ'_2 that belong to the set of states S after the loop. From the assumption (F-OX), we get the existence of two states σ_1 and σ_2 from S_p, such that $\sigma_1(x) = \sigma'_1(x)$ and $\sigma_2(x) = \sigma'_2(x)$. Because of the precondition, we know that $\sigma_1(x) = \sigma_2(x)$, and thus can conclude that $\sigma'_1(x) = \sigma'_2(x)$.

Framing hyperproperties with existentially-quantified states. Note that the rule *FrameSafe* has the restriction that F is not allowed to existentially quantify over states. To see why this is a limitation, consider as an example the method `framing2` on the right of Fig. 12. This method has the same body and loop invariant as method `framing1`, but we now want to prove that if there is an execution whose initial value x is maximal (among all executions), then there should exist an execution whose final value for y is also maximal. In this case, we would like to apply the rule *FrameSafe* with the frame $F \triangleq (\exists \langle \sigma \rangle. \forall \langle \sigma' \rangle. \sigma(x) \geq \sigma'(x))$, but cannot because of this restriction.

To overcome this limitation, Hyper Hoare Logic provides the following rule, which lifts this restriction:

$$\frac{\mathit{mod}(C) \cap \mathit{fv}(F) = \varnothing \quad \models_{\Downarrow} [P]\ C\ [Q] \quad F \text{ is a syntactic hyper-assertion}}{\models_{\Downarrow} [P \wedge F]\ C\ [Q \wedge F]}\ (\mathit{Frame})$$

This rule requires however to prove a stronger triple, the *terminating* hyper-triple $\models_{\Downarrow} [P]\ C\ [Q]$, which must ensure the existence of a terminating execution from any initial state. In our tool, we can ensure that a triple around a loop **while** (b) $\{C\}$ is terminating as long as C contains no **`assume`** statements, and this loop and all nested loops in C terminate, i.e., have been annotated with a **`decreases`** clause. This is for example the case for the loop in method `framing2`. When those two conditions hold, it is sound to strengthen the previous encoding with the additional assumption (F-UX), as follows:

```
S_p := S
... // loop encoding
assume ∀σ' ∈ S. ∃σ ∈ S_p. (∀x. x ∉ fv(C) ⇒ σ(x) = σ'(x)) // (F-OX)
assume ∀σ ∈ S_p. ∃σ ∈ S. (∀x. x ∉ fv(C) ⇒ σ(x) = σ'(x)) // (F-UX)
```

Together, those two assumptions are stronger than the application of the rule *Frame* for any frame F. For example, emitting those two assumptions together lets us automatically derive that $\exists \langle \sigma \rangle. \forall \langle \sigma' \rangle. \sigma(x) \geq \sigma'(x)$ holds after the loop in our example. However, we have noticed in practice that emitting the assumption (F-UX) does not interact well with the encoding described in Sect. 3.2,

and might result in matching loops. Thus, our tool provides an option to emit this second assumption (when applicable), which is disabled by default.

5 Implementation and Evaluation

We implemented Hypra, a deductive program verifier for hyperproperties, on top of Viper; that is, Hypra takes as input a text file, translates it into a Viper program (as described in Sect. 3), calls the Viper verifier to verify this program, and then translates the output (successful verification or error messages) back to the user.[9]

We evaluated Hypra on a diverse set of examples, which includes many examples from the literature, to answer the following questions:
(RQ1) Can our tool (dis-)prove hyperproperties of different types, namely $\forall^*$, $\exists^*$, $\forall^*\exists^*$, and $\exists^*\forall^*$?
(RQ2) How many lines of proof annotations are needed by our tool?
(RQ3) Can our tool verify complex examples in a reasonable amount of time?

In summary, our evaluation shows that our tool can efficiently (dis-)prove hyperproperties of different types with a reasonable amount of proof annotations, and it can do so within a reasonable amount of time. In the following, we describe how we selected our benchmarks, how we ran the experiments, and present and discuss the results.

Benchmarks. To evaluate Hypra, we used the benchmarks from Descartes [Sousa and Dillig 2016], HyPa [Beutner and Finkbeiner 2022], ORHLE [Dickerson et al. 2022], and PCSat [Unno et al. 2021]: We selected a subset of their publicly-available benchmarks and translated them into the programming language supported by our tool to form our benchmarks. We selected the benchmarks based on the following criteria:

(1) For Descartes, ORHLE, HyPa, and PCSat, we ignored the benchmarks that use data structures not supported by our tool, such as arrays.
(2) For Descartes, we ignored the benchmarks that use objects with more than 3 fields, since translating fields into the language supported by our tool is cumbersome.
(3) For ORHLE, HyPa, and PCSat, we ignored the benchmarks that prove hyperproperties over *different* programs, since our tool does not support this.
(4) For PCSat, we selected only the benchmarks that do not require reasoning about co-termination, since our tool does not support this.

For each selected benchmark, we translated it to the syntax accepted by Hypra. To obtain hyper-triples semantically equivalent to the original specifications, we used the formal translations given by Dardinier and Müller [2024]. In addition, we annotated the translated benchmarks with loop variants, loop invariants and hints when necessary.

For invalid benchmarks that fail to prove $\forall^*$ or $\forall^*\exists^*$-hyperproperties, we also formally disprove them. To do so, we strengthened the preconditions and proved the negation of the original postconditions [Dardinier and Müller 2024, Theorem 4]. In particular, this allows us to obtain benchmarks that prove $\exists^*\forall^*$-hyperproperties, which are not included in the benchmark suites we draw from.

In total, we obtained 84 benchmarks. Fig. 13 provides more details about the selected benchmarks.

Experimental Setup. We ran Hypra to verify the translated benchmarks on a MacBook Pro running macOS Ventura 13.3 with a 2.3 GHz 8-Core Intel Core i9 processor and 32 GB RAM. Each benchmark was run with 10 repetitions. For each run, we recorded the verification result and runtime. In the end, we checked that the verification results in all runs were consistent, and also computed the average verification time for each benchmark.

[9] Viper actually provides two verifiers, one based on symbolic execution, and one based on Boogie [Leino 2008]. Our tool uses the two Viper verifiers to verify the generated Viper program, and reports the first successful verification result.

Type of hyperproperty	Source	Files		Verification time		Annotations
		no.	Mean (LoC)	Mean (s)	Median (s)	Mean (LoC)
$\forall^*$	Descartes	15	129	2.3	1.7	0.0
	PCSat	3	23	1.1	1.1	2.7
	Overall	**18**	**111**	**2.1**	**1.6**	**0.4**
$\exists^*$	Descartes†	8	81	13.0	5.0	0.0
	ORHLE	6	29	2.9	2.5	7.7
	Overall	**14**	**59**	**8.7**	**3.5**	**3.3**
$\forall^*\exists^*$	ORHLE	28	20	2.3	1.4	1.2
	HyPa	8	14	1.2	1.1	2.1
	PCSat	1	22	1.2	1.2	2.0
	Overall	**37**	**19**	**2.0**	**1.3**	**1.4**
$\exists^*\forall^*$	ORHLE†	15	25	1.6	1.2	1.7

Fig. 13. 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.

Results. The results of our evaluation are shown in Fig. 13. As we can see, Hypra can handle not only all $\forall^*$, $\exists^*$ and $\forall^*\exists^*$-hyperproperties that other verifiers can handle, but also $\exists^*\forall^*$-hyperproperties, which no other existing verifier supports.

Although verification using Hypra is not fully automated, it only requires a reasonable amount of proof annotations from users, which is evidenced by the last column of Fig. 13.

Moreover, Hypra is quite efficient in general. On average, it took Hypra 258 seconds to run the entire benchmark suite composed of 84 programs. For 93% of the benchmarks, verification finished within 5 seconds. In some rare cases, the runtime was relatively long, with the maximum runtime around 35 seconds. This is not unexpected, since some of those benchmarks have very complex commands (such as lots of nested conditional statements) and specifications (such as preconditions and postconditions of the shape $\exists\exists\exists\forall\forall\forall$).

In summary, our evaluation demonstrates that Hypra can effectively verify hyperproperties of different types with a reasonable amount of proof annotations and within a reasonable amount of time.

6 Related Work

In this section, we first cover related program logics for hyperproperties (Sect. 6.1), and then tools and approaches for automatically verifying hyperproperties (Sect. 6.2).

6.1 Program Logics for Hyperproperties

As discussed in Sect. 1, many logics to prove that a program satisfies a safety hyperproperty [Clarkson and Schneider 2008] have been proposed over the years [Naumann and Ngo 2019]. Many of those logics actually prove *relational* properties, i.e., properties that relate the executions of several (potentially different) programs. For example, Relational Hoare Logic (RHL) [Benton 2004] extends Hoare Logic [Floyd 1967; Hoare 1969] to reason about $\forall\forall$-properties relating the executions of two programs, e.g., to prove the correctness of some program transformations. RHL has later been extended to handle more complex programs. For example, RHL has been combined with separation logic [Reynolds 2002], to support relational properties between two heap-manipulating

programs in a modular way [Yang 2007]. RHL has also been extended to reason about higher-order programs [Aguirre et al. 2017]. Several program logics [Amtoft et al. 2006; Costanzo and Shao 2014; Eilers et al. 2023; Ernst and Murray 2019] have been designed specifically to prove non-interference [Volpano et al. 1996] properties, a particular case of 2-safety hyperproperties over pairs of executions of a *single* program.

Several important safety hyperproperties are not 2-safety hyperproperties, but k-safety hyperproperties for $k > 2$, such as transivity ($k = 3$) or associativity ($k = 4$). Sousa and Dillig [2016] have proposed Cartesian Hoare Logic (CHL) to reason about k-safety properties for any fixed k. D'Osualdo et al. [2022] have identified several limitations of CHL when trying to compose together proofs of different k-safety properties, and have proposed a novel weakest-precondition calculus to overcome these limitations. Gladshtein et al. [2024] have further extended the previous calculus to handle heap-manipulating programs and k-safety properties for values of k depending on program variables, which is useful to specify and verify computations over structured data.

All aforementioned logics are *overapproximate* logics, that is, they work by overapproximating the set of executions, which is sufficient for proving safety hyperproperties. However, hyperproperties outside the safety class require proving the *existence* of relevant executions, which requires *underapproximation*. Underapproximate logics include Reverse Hoare Logic [de Vries and Koutavas 2011] and Incorrectness Logic [O'Hearn 2019], which are useful to prove reachability properties or the existence of bugs. Underapproximate logics have proven useful to justify the formal foundations of industrial bug-finding tools [Blackshear et al. 2018; Distefano et al. 2019; Gorogiannis et al. 2019; Le et al. 2022]. Several logics combine over- and underapproximation reasoning for *single* executions, such as Dynamic Logic [Harel 1979], Outcome Logic [Zilberstein et al. 2023], Exact Separation Logic [Maksimović et al. 2023], or Local Completeness Logic [Bruni et al. 2021]. Recently, Murray [2020] has proposed a program logic for $\exists\exists$-hyperproperties, to prove the presence of insecurity in a program.

Finally, several program logics combining over- and underapproximation reasoning for relating *multiple* executions have been proposed, to reason about $\forall^*\exists^*$ or $\exists^*\forall$-hyperproperties. Maillard et al. [2019] present a general framework for defining relational program logics (for two executions of two potentially different programs), which can be instantiated for $\forall\exists$-properties. RHLE [Dickerson et al. 2022] combines an underapproximate and an overapproximate Hoare logic, to support relational $\forall^*\exists^*$-properties between (potentially different) programs. Antonopoulos et al. [2023] present BiKAT, an extension of KAT [Kozen 1997], useful to reason about alignment in the context of relational verification, and derive from BiKAT inference rules for $\forall\forall$ and $\forall\exists$-properties. The authors also show that BiKAT can in principle also be used for $\exists\exists$ and $\exists\forall$-properties. Finally, Dardinier and Müller [2024] present Hyper Hoare Logic (HHL), on which we base our work. Unlike the previously-mentioned logics, HHL is tailored to properties relating the executions of a *single* program, and does not support relational properties between different programs. However, HHL supports a larger class of hyperproperties than these logics, since it supports hyperproperties with arbitrary quantifier alternations.

6.2 Automated Verification of Hyperproperties

Deductive Verification. Deductive verifiers are tools that, given as input a program, a specification, and proof hints (such as loop invariants), try to automatically construct a proof in a given program logic that the program satisfies the specification. Many deductive verifiers based on SMT solvers (such as Z3 [de Moura and Bjørner 2008]) have been developed for verifying *safety properties*, i.e., properties that should hold for all *individual* executions, such as Boogie [Leino 2008], Why3 [Filliâtre and Paskevich 2013], Dafny [Leino 2010], or Viper [Müller et al. 2016].

The problem of verifying that a program satisfies a k-safety hyperproperty can be reduced to the problem of verifying that a product program [Barthe et al. 2011; Terauchi and Aiken 2005] satisfies a safety property, where the product program is for example obtained by composing sequentially k renamed copies of the original program. The product program can then be verified using deductive verifiers tailored for safety properties. Eilers et al. [2019] show how to treat method calls modularly in this context, allowing methods to have relational preconditions and postconditions, similar to the $\forall^*$-specifications shown in Sect. 2 (for example in Fig. 2).

Deductive verifiers specifically targeting hyperproperties have been developed as well. Those include WhyRel [Nagasamudram et al. 2023], SecC (based on SecCSL) [Ernst and Murray 2019], HyperViper (based on CommCSL) [Eilers et al. 2023] for non-interference [Volpano et al. 1996] (a 2-safety hyperproperty), Descartes (based on Cartesian Hoare Logic) [Sousa and Dillig 2016] for k-safety hyperproperties, and ORHLE (based on RHLE) [Dickerson et al. 2022] for $\forall^*\exists^*$-hyperproperties. As our evaluation shows, our tool handles well the benchmarks from Descartes and ORHLE, and can even disprove invalid ones. Compared to ORHLE, the closest to our work, our tool Hypra is more expressive, since it also supports for example $\exists^*\forall^*$-hyperproperties, and supports reasoning about runtime errors. Our tool is also more flexible, since it allows the user to write explicit quantifiers in the assertion language itself, and thus allows one to combine different types of hyperproperties in the same proof, whereas ORHLE requires the user to fixed the quantification scheme in advance. Moreover, even for $\forall^*\exists^*$-hyperproperties, our tool supports reasoning about more complex proof patterns, such as while loops where different executions might exit at different iterations.

Other approaches have been developed to automatically verify hyperproperties [Assaf et al. 2017; Barthe et al. 2019; Farzan and Vandikas 2019; Itzhaky et al. 2024; Unno et al. 2021]. For example, Assaf et al. [2017] use abstract interpretation [Cousot and Cousot 1977] to verify different hypersafety properties related to information flow, including some safety hyperproperties that are not k-safety for any k. To achieve this, they present a hypercollecting semantics, similar in spirit to the function *sem* (Def. 1) from Hyper Hoare Logic. Unno et al. [2021] present PCSat, a tool based on a generalization of Constrained Horn Clauses [Bjørner et al. 2015] to automatically verify k-safety hyperproperties, and more complex hyperproperties such as termination-sensitive non-interference [Volpano and Smith 1997], and generalized non-interference [McCullough 1987; McLean 1996]. As shown in our evaluation, our tool Hypra can handle all the benchmarks from PCSat that fall in our supported subset of programs, with a reasonable amount of proof annotations and in reasonable time. Extending Hypra to reason about properties such as termination-sensitive non-interference is future work.

Finally, *temporal* logics to express hyperproperties have been proposed, such as HyperLTL and HyperCTL* [Clarkson et al. 2014], and model checking [Clarke 1997] algorithms to check whether finite-state systems satisfy hyperproperties expressed in these temporal logics have been proposed [Finkbeiner et al. 2015]. For example, Hsu et al. [2021] have proposed algorithms for *bounded* model checking, Coenen et al. [2019] proposed model checking algorithms for $\forall^*\exists^*$-hyperproperties, and Beutner and Finkbeiner [2023] proposed an explicit-state model checking algorithm that is complete for HyperLTL and for hyperproperties with arbitrary quantifier alternations. Beutner and Finkbeiner [2022] have also shown that model checking techniques for $\forall^*\exists^*$ can be applied to infinite-state systems, by using predicate abstraction.

7 Conclusion and Future Work

In this paper, we have presented a novel approach for the deductive verification of hyperproperties, including hyperproperties of the shape $\forall^*\exists^*$ and $\exists^*\forall^*$. Our approach, based on an extension of Hyper Hoare Logic [Dardinier and Müller 2024] to reason about runtime errors (Sect. 2.2) and

implemented in the tool Hypra, tracks a lower bound and an upper bound of the set of reachable states and the set of error states (Sect. 3.1), and combines both bounds with some carefully-designed restrictions to avoid matching loops (Sect. 3.2). Our tool is able to automatically generate verification conditions for loops, based on a user-provided loop invariant and an optional variant. To achieve this, our tool automatically selects the best loop rule to apply based on the context (Sect. 4.1), which includes a novel loop rule for proving $\forall^*\exists^*$-hyperproperties (Sect. 4.2), proved sound in Isabelle/HOL. Finally, our tool is able to automatically frame hyperproperties untouched by loops (Sect. 4.3), leading to more concise loop invariants. Our evaluation (Sect. 5) shows that Hypra can prove a large class of hyperproperties for a large class of programs, in a reasonable amount of time and with a reasonable amount of proof annotations.

Our work opens several avenues for future work. Currently, our tool does not support heap-manipulating programs. One interesting research direction would thus be to extend Hyper Hoare Logic to support reasoning about heap-manipulating programs in a modular way (for example by borrowing concepts from separation logic [Reynolds 2002]), and then extend our approach based on this extended logic. Another interesting research direction would be to extend our approach to support relational properties with arbitrary quantifier alternations between *different* programs, which could for example allow one to prove that a program *does not* refine another one, an $\exists\forall$-property. Finally, similar to how we extended Hyper Hoare Logic to support runtime errors, it would be interesting to explore an extension of Hyper Hoare Logic to reason about termination and non-termination *in the assertion language itself*. On top of proving termination (which our tool is already capable of), this would enable proving non-termination, and hyperproperties such as co-termination (e.g., to prove that observing non-termination does not leak any secret information).

Acknowledgments

This work was partially funded by the Swiss National Science Foundation (SNSF) under Grant No. 197065.

Data Availability Statement

Our artifact [Dardinier et al. 2024] consists of (1) our tool Hypra, (2) our evaluation, with instructions on how to reproduce the results, and (3) Isabelle/HOL proofs of the soundness of the novel loop rule described in Sect. 4.1 (Thm. 1) and of Lemma 1.

References

Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2017. A relational logic for higher-order programs. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–29.

Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. 2006. A Logic for Information Flow in Object-Oriented Programs. *SIGPLAN Not.* 41, 1 (jan 2006), 91–102. https://doi.org/10.1145/1111320.1111046

Timos Antonopoulos, Eric Koskinen, Ton Chanh Le, Ramana Nagasamudram, David A. Naumann, and Minh Ngo. 2023. An Algebra of Alignment for Relational Verification. *Proc. ACM Program. Lang.* 7, POPL, Article 20 (jan 2023), 31 pages. https://doi.org/10.1145/3571213

Mounir Assaf, David A Naumann, Julien Signoles, Eric Totel, and Frédéric Tronel. 2017. Hypercollecting semantics and its application to static analysis of information flow. *ACM SIGPLAN Notices* 52, 1 (2017), 874–887.

Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Mathematical Structures in Computer Science* 21, 6 (2011), 1207–1252.

Gilles Barthe, Renate Eilers, Pamina Georgiou, Bernhard Gleiss, Laura Kovács, and Matteo Maffei. 2019. Verifying relational properties using trace logic. In *2019 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 170–178.

Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 14–25. https://doi.org/10.1145/964001.964003

Raven Beutner and Bernd Finkbeiner. 2022. Software Verification of Hyperproperties Beyond k-Safety. In *Computer Aided Verification*, Sharon Shoham and Yakir Vizel (Eds.). Cham, 341–362.

Raven Beutner and Bernd Finkbeiner. 2023. AutoHyper: Explicit-State Model Checking for HyperLTL. In *Tools and Algorithms for the Construction and Analysis of Systems*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer Nature Switzerland, Cham, 145–163.

Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. *Horn Clause Solvers for Program Verification.* Springer International Publishing, Cham, 24–51. https://doi.org/10.1007/978-3-319-23534-9_2

Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (oct 2018), 28 pages. https://doi.org/10.1145/3276514

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2021. A Logic for Locally Complete Abstract Interpretations. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS).* 1–13. https://doi.org/10.1109/LICS52264.2021.9470608

Edmund M Clarke. 1997. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17.* Springer, 54–56.

Michael R Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K Micinski, Markus N Rabe, and César Sánchez. 2014. Temporal logics for hyperproperties. In *International Conference on Principles of Security and Trust.* 265–284.

Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *21st IEEE Computer Security Foundations Symposium.* 51–65. https://doi.org/10.1109/CSF.2008.7

Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. 2019. Verifying hyperliveness. In *International Conference on Computer Aided Verification.* 121–139.

David Costanzo and Zhong Shao. 2014. A Separation Logic for Enforcing Declarative Information Flow Control Policies. In *Principles of Security and Trust*, Martín Abadi and Steve Kremer (Eds.). 179–198.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* 238–252.

Thibault Dardinier, Anqi Li, and Peter Müller. 2024. *Hypra: A Deductive Program Verifier for Hyperproperties (artifact).* https://doi.org/10.5281/zenodo.12671562

Thibault Dardinier and Peter Müller. 2024. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties. *Proc. ACM Program. Lang.* 8, PLDI, Article 207 (jun 2024), 25 pages. https://doi.org/10.1145/3656437

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). 337–340.

Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare Logic. In *Software Engineering and Formal Methods*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.). 155–171.

David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: a theorem prover for program checking. *J. ACM* 52, 3 (2005), 365–473. https://doi.org/10.1145/1066100.1066102

Robert Dickerson, Qianchuan Ye, Michael K. Zhang, and Benjamin Delaware. 2022. RHLE: Modular Deductive Verification of Relational ∀∃ Properties. In *Programming Languages and Systems: 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings* (Auckland, New Zealand). 67–87. https://doi.org/10.1007/978-3-031-21037-2_4

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (jul 2019), 62–70. https://doi.org/10.1145/3338112

Emanuele D'Osualdo, Azadeh Farzan, and Derek Dreyer. 2022. Proving Hypersafety Compositionally. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 135 (oct 2022), 26 pages. https://doi.org/10.1145/3563298

Marco Eilers, Thibault Dardinier, and Peter Müller. 2023. CommCSL: Proving Information Flow Security for Concurrent Programs Using Abstract Commutativity. *Proc. ACM Program. Lang.* 7, PLDI, Article 175 (jun 2023), 26 pages. https://doi.org/10.1145/3591289

Marco Eilers, Peter Müller, and Samuel Hitz. 2019. Modular product programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42, 1 (2019), 1–37.

Gidon Ernst and Toby Murray. 2019. SecCSL: Security Concurrent Separation Logic. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Cham, 208–230.

Azadeh Farzan and Anthony Vandikas. 2019. Automated Hypersafety Verification. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 200–218.

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—where programs meet provers. In *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22.* Springer, 125–128.

Bernd Finkbeiner, Markus N Rabe, and César Sánchez. 2015. Algorithms for model checking HyperLTL and HyperCTL. In *International Conference on Computer Aided Verification.* Springer, 30–48.

Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium in Applied Mathematics* (1967), 19–32.

Vladimir Gladshtein, Qiyuan Zhao, Willow Ahrens, Saman Amarasinghe, and Ilya Sergey. 2024. Mechanised Hypersafety Proofs about Structured Data. *Proc. ACM Program. Lang.* 8, PLDI, Article 173 (jun 2024), 24 pages. https://doi.org/10.

1145/3656403

Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2019. A True Positives Theorem for a Static Race Detector. *Proc. ACM Program. Lang.* 3, POPL, Article 57 (jan 2019), 29 pages. https://doi.org/10.1145/3290370

David Harel. 1979. *First-order dynamic logic.* Springer.

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (oct 1969), 576–580. https://doi.org/10.1145/363235.363259

Tzu-Han Hsu, César Sánchez, and Borzoo Bonakdarpour. 2021. Bounded Model Checking for Hyperproperties. In *Tools and Algorithms for the Construction and Analysis of Systems*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer International Publishing, Cham, 94–112.

Shachar Itzhaky, Sharon Shoham, and Yakir Vizel. 2024. Hyperproperty Verification as CHC Satisfiability. arXiv:2304.12588 [cs.LO]

Dexter Kozen. 1997. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.* 19, 3 (may 1997), 427–443. https://doi.org/10.1145/256167.256195

Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 81 (apr 2022), 27 pages. https://doi.org/10.1145/3527325

K. Rustan M. Leino. 2008. This is Boogie 2. (June 2008). https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*. Springer, 348–370.

K. Rustan M. Leino and Rosemary Monahan. 2009. Reasoning about comprehensions with first-order SMT solvers. In *Proceedings of the 2009 ACM Symposium on Applied Computing* (Honolulu, Hawaii) *(SAC '09)*. Association for Computing Machinery, New York, NY, USA, 615–622. https://doi.org/10.1145/1529282.1529411

Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. 2019. The next 700 Relational Program Logics. *Proc. ACM Program. Lang.* 4, POPL, Article 4 (dec 2019), 33 pages. https://doi.org/10.1145/3371072

Petar Maksimović, Caroline Cronjäger, Andreas Lööw, Julian Sutherland, and Philippa Gardner. 2023. Exact Separation Logic: Towards Bridging the Gap Between Verification and Bug-Finding. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, Vol. 263. 19:1–19:27. https://doi.org/10.4230/LIPIcs.ECOOP.2023.19

Daryl McCullough. 1987. Specifications for multi-level security and a hook-up. In *1987 IEEE Symposium on Security and Privacy*. IEEE, 161–161.

John McLean. 1996. A general theory of composition for a class of" possibilistic" properties. *IEEE Transactions on Software Engineering* 22, 1 (1996), 53–67.

P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI) (LNCS, Vol. 9583)*, B. Jobstmann and K. R. M. Leino (Eds.). Springer-Verlag, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Toby Murray. 2020. An Under-Approximate Relational Logic: Heralding Logics of Insecurity, Incorrect Implementation and More. https://doi.org/10.48550/ARXIV.2003.04791

Ramana Nagasamudram, Anindya Banerjee, and David A. Naumann. 2023. The WhyRel Prototype for Modular Relational Verification of Pointer Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer Nature Switzerland, Cham, 133–151.

David A. Naumann and Minh Ngo. 2019. Whither Specifications as Programs. In *Unifying Theories of Programming*, Pedro Ribeiro and Augusto Sampaio (Eds.). Springer International Publishing, Cham, 39–61.

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer-Verlag, Berlin, Heidelberg.

Peter W. O'Hearn. 2019. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (dec 2019), 32 pages. https://doi.org/10.1145/3371078

J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

Marcelo Sousa and Isil Dillig. 2016. Cartesian Hoare Logic for Verifying K-Safety Properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 57–69. https://doi.org/10.1145/2908080.2908092

Tachio Terauchi and Alex Aiken. 2005. Secure information flow as a safety problem. In *International Static Analysis Symposium*. 352–367.

Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 2021. Constraint-Based Relational Verification. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 742–766.

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A sound type system for secure flow analysis. *Journal of computer security* 4, 2-3 (1996), 167–187.

D. Volpano and G. Smith. 1997. Eliminating covert flows with minimum typings. In *Proceedings 10th Computer Security Foundations Workshop*. 156–168. https://doi.org/10.1109/CSFW.1997.596807

Hongseok Yang. 2007. Relational separation logic. *Theoretical Computer Science* 375, 1 (2007), 308–334. https://doi.org/10.1016/j.tcs.2006.12.036

Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 2023. Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. https://www.cs.cornell.edu/~noamz/files/pubs/outcome.pdf

Received 2024-04-06; accepted 2024-08-18