Automated Software Verification of Hyperliveness

Raven Beutner

CISPA Helmholtz Center for Information Security, Germany
raven.beutner@cispa.de

Abstract. Hyperproperties relate multiple executions of a program and are commonly used to specify security and information-flow policies. Most existing work has focused on the verification of k-safety properties, i.e., properties that state that *all* k-tuples of execution traces satisfy a given property. In this paper, we study the automated verification of richer properties that combine universal and existential quantification over executions. Concretely, we consider $\forall^k\exists^l$ properties, which state that for all k executions, there exist l executions that, together, satisfy a property. This captures important non-k-safety requirements, including hyperliveness properties such as generalized non-interference, opacity, refinement, and robustness. We design an automated constraint-based algorithm for the verification of $\forall^k\exists^l$ properties. Our algorithm leverages a sound-and-complete program logic and a (parameterized) strongest postcondition computation. We implement our algorithm in a tool called `ForEx` and report on encouraging experimental results.

Keywords: Hyperproperties · Program Logic · Hoare Logic · Symbolic Execution · Constraint-based Verification · Predicate Transformer · Refinement · Strongest Postcondition · Underapproximation.

1 Introduction

Relational properties (also called hyperproperties [21]) move away from a traditional specification that considers all executions of a system in isolation and, instead, relate *multiple* executions. Hyperproperties are becoming increasingly important and have shown up in various disciplines, perhaps most prominently in information-flow control. Assume we are given a program $\mathbb{P}$ with high-security input h, low-security input l, and public output o, and we want to formally prove that the output of $\mathbb{P}$ does not leak information about h. One way to ensure this is to verify that $\mathbb{P}$ behaves deterministically in the low-security input l, i.e., if the low-security input is identical across *two* executions, so is $\mathbb{P}$'s output.

The above property is a typical example of a 2-safety property stating a requirement on all pairs of traces. More generally, a k-safety property requires that *all* k-tuples of executions, together, satisfy a given property. In the last decade, many approaches for the verification of k-safety properties have been proposed, based, e.g., on model-checking [55,33,31], abstract interpretation [43,41,4,44], symbolic execution [30], or program logics [7,56,28,60,49].

```
if (h > l) then
  o = l + ⋆ℕ
else
  x = ⋆ℕ
  if (x > l) then
    o = x
  else
    o = l
```

Fig. 1: Example program

However, for many relational properties, the implicit *universal* quantification found in k-safety properties is too restrictive. Consider the simple program in Figure 1 (taken from [12]), where $\star_{\mathbb{N}}$ denotes the nondeterministic choice of a natural number. This program clearly violates the 2-safety property discussed above as the nondeterminism influences the final value of o. Nevertheless, the program does not leak any information about the secret input h. To see this, assume the attacker observes some fixed low-security input-output pair (l, o), i.e., the attacker observes everything except the high-security input. The key observation is that (l, o) is possible for any possible high-security input, i.e., for every value of h, there *exists* some way to resolve the nondeterminism such that (l, o) is the observation made by the attacker. This information-flow policy – called generalized non-interference (GNI) [45] – requires a combination of universal and existential reasoning and thus cannot be expressed as a k-safety property.

FEHTs. In this paper, we study the automated verification of such (functional) $\forall^*\exists^*$ properties. Concretely, we consider specifications in a form we call Forall-Exist Hoare Tuples (FEHT) (also called *refinement quadruples* [5] or *RHLE triples* [26]), which have the form

$$\langle \Phi \rangle \mathbb{P}_1 \circledast \cdots \circledast \mathbb{P}_k \sim \mathbb{P}_{k+1} \circledast \cdots \circledast \mathbb{P}_{k+l} \langle \Psi \rangle,$$

where $\mathbb{P}_1, \ldots, \mathbb{P}_{k+l}$ are (possibly identical) programs and Φ, Ψ are first-order formulas that relate $k + l$ different program runs. The FEHT is valid if for *all* $k + l$ initial states that satisfy Φ, and for *all* possible executions of $\mathbb{P}_1, \ldots, \mathbb{P}_k$ there *exist* executions of $\mathbb{P}_{k+1}, \ldots, \mathbb{P}_{k+l}$ such that the final states satisfy Ψ. For example, GNI can be expressed as $\langle l_1 = l_2 \rangle \mathbb{P} \sim \mathbb{P} \langle o_1 = o_2 \rangle$, where l_1 and o_1 (resp. l_2 and o_2) refer to the value of l and o in the first (resp. second) program copy. That is, for *any* two initial states σ_1, σ_2 with identical values for l (but possibly different values for h), and *any* final state σ_1' reachable by executing $\mathbb{P}$ from σ_1, there *exists* some final state σ_2' (reachable from σ_2 by executing $\mathbb{P}$) that agrees with σ_1' in the value of o. The program in Figure 1 satisfies this FEHT. In the terminology of Clarkson and Schneider [21], GNI is a *hyperliveness* property, hence the name of our paper. Intuitively, the term hyperliveness stems from the fact that – due to the existential quantification in FEHTs – GNI reasons about the existence of a particular execution. Similar to the definition of liveness in temporal properties [2], we can, therefore, satisfy GNI by *adding* sufficiently many execution traces [22].

Verification Using a Program Logic. For *finite*-state hardware systems, many automated verification methods for hyperliveness properties (e.g., in the form of FEHTs) have been proposed [20,38,15,33,13,14,22]. In contrast, for *infinite*-state software, the verification of FEHTs is notoriously difficult; FEHTs mix

quantification of different types, so we cannot employ purely over-approximate reasoning principles (as is possible for k-safety). Most existing approaches for software verification, therefore, require substantial user interaction, e.g., in the form of a custom Horn-clause template [57], a user-provided abstraction [12], or a deductive proof strategy [26,5]. See Section 6 for more discussion.

In this paper, we put forward an automatic algorithm for the verification of FEHTs. Our method is rooted in a novel *program logic*, which we call Forall-Exist Hoare Logic (FEHL) (in Section 3). Similar to many program logics for k-safety properties [56,19], our logic focuses on one of the programs involved in the verification at any given time (by, e.g., symbolically executing one step in one of the programs) and thus lends itself to automation. We show that FEHL is sound and complete (relative to a complete proof system for over- and under-approximate unary Hoare triples).

Automated Verification. Our verification algorithm – presented in Section 4 – then leverages FEHL for the analysis of FEHTs. During this analysis, the key algorithmic challenge is to find suitable instantiations for nondeterministic choices made in existentially quantified executions. Our algorithm avoids a direct instantiation and instead treats the outcome of the nondeterministic choice *symbolically*, allowing an instantiation at a later point in time. Formally, we define the concept of a *parametric assertion*. Instead of capturing a set of states, a parametric assertion defines a function that maps concrete values for a set of parameters (in our case, the nondeterministic choices in existentially quantified programs whose concrete instantiations we have postponed) to sets of states. Our algorithm then recursively computes a *parametric postcondition* and delegates the search for appropriate instantiations of the parameters to an SMT solver. Crucially, our algorithm only explores a restricted class of program alignments (as guided by FEHL). Therefore, the resulting constraints are ordinary (first-order) SMT formulas, which can be handled using off-the-shelf SMT solvers.

Implementation and Experiments. We implement our algorithm in a tool called `ForEx` and compare it with existing approaches for the verification of $\forall^*\exists^*$ properties (in Section 5). As `ForEx` can resort to highly optimized off-the-shelf SMT solvers, it outperforms existing approaches (which often rely on custom solving strategies) in many benchmarks.

2 Preliminaries

Programs. Let $\mathcal{V}$ be a set of program variables. We consider a simple (integer-valued) programming language generated by the following grammar.

$$\mathbb{P}, \mathbb{Q} := \texttt{skip} \mid x = e \mid \texttt{assume}(b) \mid \texttt{if}(b, \mathbb{P}, \mathbb{Q}) \mid \texttt{while}(b, \mathbb{P}) \mid \mathbb{P}\,;\,\mathbb{Q} \mid x = \star$$

where $x \in \mathcal{V}$ is a variable, e is a (deterministic) arithmetic expressions over variables in $\mathcal{V}$, and b is a (deterministic) boolean expression. `skip` denotes the program that does nothing; $x = e$ assigns x the result of evaluating e; `assume`(b)

assumes that b holds, i.e., does not continue execution from states that do not satisfy b; $\texttt{if}(b, \mathbb{P}, \mathbb{Q})$ executes $\mathbb{P}$ if b holds and otherwise executes $\mathbb{Q}$; $\texttt{while}(b, \mathbb{P})$ executes $\mathbb{P}$ as long as b holds; $\mathbb{P} \fatsemi \mathbb{Q}$ executes $\mathbb{P}$ followed by $\mathbb{Q}$; and $x = \star$ assigns x some nondeterministically chosen integer. For an arithmetic expression e, we write $\mathit{Vars}(e) \subseteq \mathcal{V}$ for the set of all variables used in the expression.

We endow our language with a standard operational semantics operating on states $\sigma : \mathcal{V} \to \mathbb{Z}$. Given a program $\mathbb{P}$, we write $[\![\mathbb{P}]\!](\sigma, \sigma')$ whenever $\mathbb{P}$ – when executed from state σ – *can* terminate in state σ'. Our semantics is defined as expected, and we give a full definition in the full version [10].

Given program states $\sigma_1 : \mathcal{V} \to \mathbb{Z}$ and $\sigma_2 : \mathcal{V}' \to \mathbb{Z}$ with $\mathcal{V} \cap \mathcal{V}' = \emptyset$, we write $\sigma_1 \oplus \sigma_2 : (\mathcal{V} \cup \mathcal{V}') \to \mathbb{Z}$ for the combined state, that behaves as σ_1 on $\mathcal{V}$ and as σ_2 on $\mathcal{V}'$. For $i \in \mathbb{N}$, we define $\mathcal{V}_i := \{x_i \mid x \in \mathcal{V}\}$ as a set of indexed program variables.

Assertions. An assertion Φ is a first-order formula over variables in $\mathcal{V}$ (or in the relational setting over $\bigcup_{i=1}^{k} \mathcal{V}_i$ for some k). Given a state σ, we write $\sigma \models \Phi$ if σ satisfies Φ. We assume that assertions stem from an arbitrarily expressive background theory such that every set of states can be expressed as a formula. This allows us to sidestep the issue of *expressiveness* in the sense of Cook [23] (see, e.g., [50,60,56] for similar treatments).

Hyperliveness Specifications. Our verification algorithm targets specifications that combine universal and existential quantification, similar to *RHLE triples* [26] and *refinement quadruples* [5]:

Definition 1. *A Forall-Exist Hoare Tuple (FEHT) has the form*

$$\langle \Phi \rangle \mathbb{P}_1 \circledast \cdots \circledast \mathbb{P}_k \sim \mathbb{P}_{k+1} \circledast \cdots \circledast \mathbb{P}_{k+l} \langle \Psi \rangle,$$

where Φ, Ψ are assertions over $\bigcup_{i=1}^{k+l} \mathcal{V}_i$, and $\mathbb{P}_1, \dots, \mathbb{P}_{k+l}$ are programs over variables $\mathcal{V}_1, \dots, \mathcal{V}_{k+l}$, respectively. The FEHT is valid *if for all states $\sigma_1, \dots, \sigma_{k+l}$ (with domains $\mathcal{V}_1, \dots, \mathcal{V}_{k+l}$, respectively) and $\sigma'_1, \dots, \sigma'_k$ such that $\bigoplus_{i=1}^{k+l} \sigma_i \models \Phi$ and $[\![\mathbb{P}_i]\!](\sigma_i, \sigma'_i)$ for all $i \in [1, k]$, there exist states $\sigma'_{k+1}, \dots, \sigma'_{k+l}$ such that $[\![\mathbb{P}_i]\!](\sigma_i, \sigma'_i)$ for all $i \in [k+1, k+l]$ and $\bigoplus_{i=1}^{k+l} \sigma'_i \models \Psi$.*

That is, we quantify universally over initial states for all $k + l$ programs (under the assumption that they, together, satisfy Φ) and also universally over executions of $\mathbb{P}_1, \dots, \mathbb{P}_k$. Afterward, we quantify *existentially* over executions of $\mathbb{P}_{k+1}, \dots, \mathbb{P}_{k+l}$ and require that the final states of all $k + l$ executions, together, satisfy the postcondition Ψ. A relational property usually refers to $k+l$ executions of the *same* program $\mathbb{P}$ (operating on variables in $\mathcal{V}$); we can model this by using α-renamed copies $\mathbb{P}_{\langle 1 \rangle}, \dots, \mathbb{P}_{\langle k+l \rangle}$ where each $\mathbb{P}_{\langle i \rangle}$ is obtained from $\mathbb{P}$ by replacing each variable $x \in \mathcal{V}$ with $x_i \in \mathcal{V}_i$. FEHTs capture a range of important properties, including e.g., non-inference [46], opacity [61], GNI [45], refinement [59], software doping [16], and robustness [18]. It is easy to see that FEHTs can also express (purely universal) k-safety properties over programs $\mathbb{P}_1, \dots, \mathbb{P}_k$ as $\langle \Phi \rangle \mathbb{P}_1 \circledast \cdots \circledast \mathbb{P}_k \sim \epsilon \langle \Psi \rangle$, where ϵ denotes the empty sequence of programs.

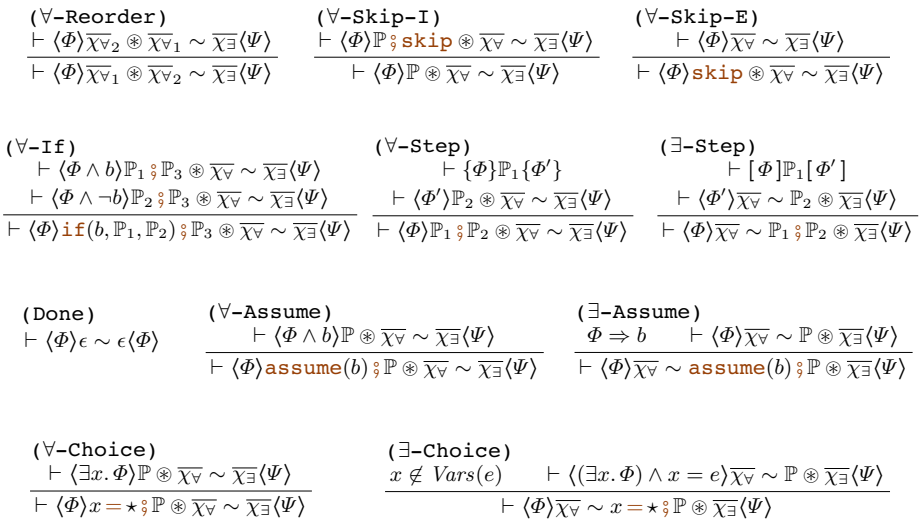


Fig. 2: Selection of core proof rules of FEHL

3 Forall-Exist Hoare Logic

The verification steps of our constraint-based algorithm (presented in Section 4) are guided by the proof rules of a novel program logic operating on FEHTs, which we call Forall-Exist Hoare Logic (FEHL).

3.1 Core Rules

We depict a selection of core rules in Figure 2; a full overview can be found in [10]. We write $\overline{\chi_\forall}$ (resp. $\overline{\chi_\exists}$) to abbreviate a list $\mathbb{P}_1 \circledast \cdots \circledast \mathbb{P}_k$ of programs that are universally (resp. existentially) quantified. Rule **(∀-Reorder)** allows for the reordering of universally quantified programs; **(∀-Skip-I)** rewrites a program $\mathbb{P}$ into $\mathbb{P}$; `skip`; **(∀-Skip-E)** removes a single `skip`-instruction; and **(Done)** derives a FEHL with an empty program sequence. Using `skip`-insertions and reordering (and the analogous rules for existentially quantified programs), we can always bring a program in the form $\mathbb{P}_1$; $\mathbb{P}_2$, targeted by the remaining rules. Rule **(∀-If)** embeds the branching condition of a conditional into the preconditions of both branches. Rules **(∀-Step)** and **(∃-Step)** allow us to resort to unary reasoning over parts of the program. These rules make the multiplicity of techniques developed for unary reasoning (e.g., symbolic execution [40] and predicate transformers [27]) applicable to the verification of hyperproperties in the form of FEHTs. For universally quantified programs of the form $\mathbb{P}_1$; $\mathbb{P}_2$, **(∀-Step)** requires an auxiliary assertion Φ' that should hold after *all* executions of $\mathbb{P}_1$ from Φ. We can express this using the standard (non-relational) Hoare triple (HT) $\{\Phi\}\mathbb{P}_1\{\Phi'\}$ [37]. The second premise then ensures that the remaining

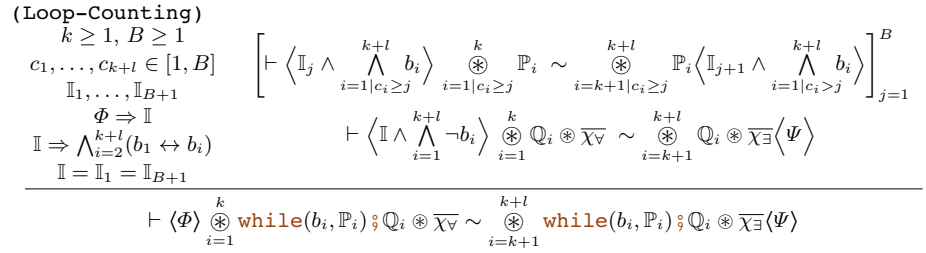


Fig. 3: Counting-based loop rule for FEHL

FEHT (after $\mathbb{P}_1$ has been executed) holds. For existentially quantified programs, we, instead, employ an underapproximation. In **(∃-Step)**, we, again, execute $\mathbb{P}_1$ but use an *Under-Approximate Hoare triple* (UHT) $[\Phi]\mathbb{P}_1[\Phi']$. The UHT $[\Phi]\mathbb{P}_1[\Phi']$ holds if for all states σ with $\sigma \models \Phi$, there *exists* a state σ' such that $[\![\mathbb{P}_1]\!](\sigma, \sigma')$ and $\sigma' \models \Phi'$.

Remark 1. UHTs behave similar to *Incorrectness Triples* (ITs) [50,58] in that they reason about the existence of a particular set of executions. The key difference is that ITs reason backward (all states in Φ' are reachable from some state in Φ), whereas UHTs reason in a forward direction (all states in Φ can reach Φ'). See, e.g., *Lisbon Triples* [47, §5] and *Outcome Triples* [62] for related approaches. We will later show that FEHL is complete when equipped with some complete proof system for UHTs (cf. Theorem 2). In the full version [10], we show that there exists at least one complete proof system for UHTs. △

For `assume` statements, **(∀-Assume)** strengthens the precondition by the assumed expression b; any state that does not satisfy b causes a (universally quantified) execution to halt and renders the FEHT vacuously valid. In contrast, **(∃-Assume)** assumes that all states in Φ satisfy b; if any state in Φ does not satisfy b, the FEHT is invalid. Likewise, the handling of a nondeterministic assignment $x = \star$ differs based on whether we consider a universally quantified or existentially quantified program. In the former case, **(∀-Choice)** removes all knowledge about the value of x within the precondition by quantifying x existentially (thus enlarging the precondition). In the latter (existentially quantified) case, we can, in a forward-style execution, choose *any* concrete value for x. **(∃-Choice)** formalizes this intuition: we first invalidate all knowledge about x and then assert that $x = e$ for some arbitrary expression e that does not depend on x. In our automated analysis (cf. Section 4), we use **(∃-Choice)**, but – instead of fixing some concrete value (or expression) at application time – we postpone the concrete instantiation by treating the value *symbolically*.

3.2 Asynchronous Loop Reasoning

A particular challenge when reasoning about relational properties is the alignment of loops. In FEHL, we propose a novel counting-based loop rule that sup-

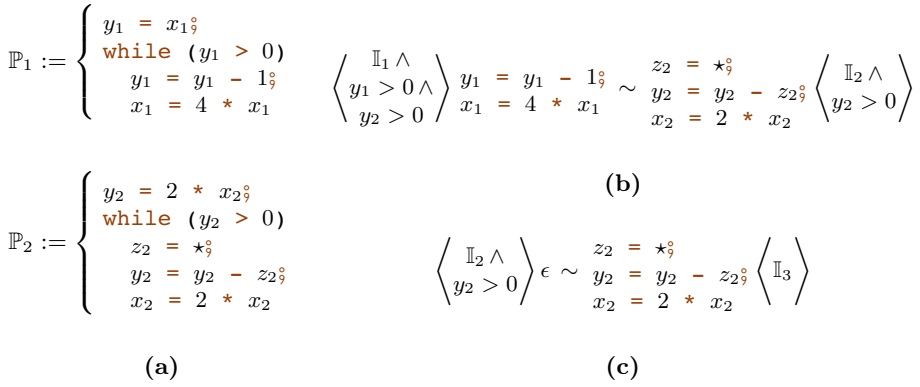


Fig. 4: In Figure 4a, we depict two example programs. In Figures 4b and 4c, we give two intermediate FEHT verification obligations (cf. Example 1).

ports asynchronous alignments while still admitting good automation. Consider the rule `(Loop-Counting)` (in Figure 3), which assumes $k \geq 1$ universally and l existentially quantified loops. The rule requires a loop invariant $\mathbb{I}$ that **(1)** is implied by the precondition ($\Phi \Rightarrow \mathbb{I}$), **(2)** ensures simultaneous termination of all loops ($\mathbb{I} \Rightarrow \bigwedge_{i=2}^{k+l}(b_1 \leftrightarrow b_i)$), and **(3)** is strong enough to establish the postcondition for the program suffixes $\mathbb{Q}_1, \ldots, \mathbb{Q}_{k+l}$ executed after the loops. The key difference from a simple synchronous traversal is that, in each "iteration", we execute the bodies of the loops for possibly different numbers of times. Concretely, `(Loop-Counting)` asks for natural numbers $c_1, \ldots, c_{k+l}$ (ranging between 1 and some arbitrary upper bound B), and – starting from the invariant $\mathbb{I}$ – we execute each $\mathbb{P}_i$ c_i times. Crucially, we need to make sure that each $\mathbb{P}_i$ will execute *at least* c_i times, i.e., the guard b_i holds after each of the first $c_i - 1$ executions. In particular, we cannot naïvely analyze c_i copies of $\mathbb{P}_i$ composed via $\fatsemi$ as this might introduce additional executions of $\mathbb{P}_i$ that would not happen in $\texttt{while}(b_i, \mathbb{P}_i)$. To ensure this, `(Loop-Counting)` demands $B+1$ intermediate assertions $\mathbb{I}_1, \ldots, \mathbb{I}_{B+1}$. In the jth iteration (for $1 \leq j \leq B$), we (symbolically) execute – from $\mathbb{I}_j$ – all loop bodies $\mathbb{P}_i$ that we want to execute at least j times (i.e., all loop bodies $\mathbb{P}_i$ where $c_i \geq j$). We require that **(1)** the postcondition $\mathbb{I}_{j+1}$ is derivable, and **(2)** the guards of all loops that we want to execute *more* than j times (i.e., loops where $c_i > j$) evaluate to true.

Example 1. Consider the two example programs $\mathbb{P}_1, \mathbb{P}_2$ in Figure 4a and the FEHT $\langle x_1 = x_2 \rangle \mathbb{P}_1 \sim \mathbb{P}_2 \langle x_1 = x_2 \rangle$. To see that this FEHT is valid, we can, in each loop iteration, always choose $z_2 = 1$. In this case, $\mathbb{P}_1$ quadruples the value of x_1 for x_1 times and $\mathbb{P}_2$ doubles the value of x_2 for $2x_2$ times, which, assuming $x_1 = x_2$, computes the same result ($x_1 = x_2 \rightarrow 4^{x_1} x_1 = 2^{2x_2} x_2$). Verifying this example automatically is challenging as both loops are executed a different number of times, so we cannot align the loops in lockstep. Likewise, computing independent (unary) summaries of both loops requires complex non-linear reasoning.

Instead, `(Loop-Counting)` enables an asynchronous alignment: After applying `(∀-Step)` and `(∃-Step)`, we are left with precondition $x_1 = x_2 \land y_2 = 2y_1$. We use `(Loop-Counting)` and align the loops such that every loop iteration in $\mathbb{P}_1$ is matched by *two* iterations in $\mathbb{P}_2$, which allows us to use a simple (linear) invariant. We set $c_1 := 1, c_2 := 2$ and define $\mathbb{I} := x_1 = x_2 \land y_2 = 2y_1$, $\mathbb{I}_1 := \mathbb{I}_3 := \mathbb{I}$, and $\mathbb{I}_2 := x_1 = 2x_2 \land y_2 = 2y_1 + 1$. Note that $\mathbb{I}$ implies the desired postcondition ($x_1 = x_2$). To establish that $\mathbb{I}$ serves as an invariant, we need to discharge the two proof obligations depicted in Figures 4b and 4c. The obligation in Figure 4b (corresponding to iteration $j = 1$) establishes that **(1)** $\mathbb{I}_2$ is a provable postcondition after executing both loop bodies from $\mathbb{I}_1$ and **(2)** that the loop in $\mathbb{P}_2$ will execute at least one more time, i.e., $y_2 > 0$. We can easily discharge this FEHT using `(∀-Step)`, `(∃-Step)`, and `(∃-Choice)` by choosing z_2 to be 1 (note that if $y_2 = 2y_1$ and $y_2 > 0$, then $y_2 - 1 > 0$). The obligation in Figure 4c corresponds to iteration $j = 2$, where we only execute the body of $\mathbb{P}_2$. We can, again, easily discharge this FEHT using `(∃-Step)` and `(∃-Choice)` (again, choosing z_2 to be 1). △

3.3 Soundness and Completeness

We can show that our proof system is sound and complete:

Theorem 1 (Soundness). *Assume that* $\vdash \{\cdot\} \cdot \{\cdot\}$ *and* $\vdash [\cdot] \cdot [\cdot]$ *are sound proof systems for HTs and UHTs, respectively. If* $\vdash \langle \Phi \rangle \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle$ *then* $\langle \Phi \rangle \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle$ *is valid.*

Theorem 2 (Completeness). *Assume that* $\vdash \{\cdot\} \cdot \{\cdot\}$ *and* $\vdash [\cdot] \cdot [\cdot]$ *are complete proof systems for HTs and UHTs, respectively. If* $\langle \Phi \rangle \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle$ *is valid then* $\vdash \langle \Phi \rangle \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle$.

Completeness follows easily by making extensive use of *unary* reasoning via (U)HTs, similar to the completeness-proof of relational Hoare logic for k-safety properties [49]. In fact, `(∀-Step)`, `(∃-Step)`, `(Done)` along with the reordering rules `(∀-Reorder)`, `(∀-Skip-I)`, and `(∀-Skip-E)` (and their analogous counterparts for existentially quantified programs) already suffice for completeness (see [10]). In the following, we leverage the *soundness* of FEHL's rules to guide our automated verification.

4 Automated Verification of Hyperliveness

Our automated verification algorithm for FEHTs follows a *strongest postcondition* computation, as is widely used in the verification of non-relational properties [1,36,51] and k-safety properties [56,19]. However, due to the inherent presence of *existential* quantification in FEHT, the strongest postcondition does, in general, not exist. For example, both $\langle \top \rangle \epsilon \sim x = \star \langle x = 1 \rangle$ and $\langle \top \rangle \epsilon \sim x = \star \langle x = 2 \rangle$ are valid but $\langle \top \rangle \epsilon \sim x = \star \langle x = 1 \land x = 2 \equiv \bot \rangle$ is clearly not. Instead, our

algorithm uses the proof rules of FEHL and treats the concrete value for nondeterministic choices in existentially quantified executions symbolically. I.e., we view the outcome as a fresh variable (called a *parameter*) that can be instantiated later. This idea of instating nondeterminism at a later point in time has already found successful application in many areas, such as existential variables in Coq or symbolic execution [40]. Our analysis brings these techniques to the realm of hyperproperty verification, which we show to yield an effective automated verification algorithm. In the following, we formally introduce parametric assertions and postconditions (in Section 4.1) and show how we can compute them using the rules of FEHL (in Sections 4.2 and 4.3).

4.1 Parametric Assertions and Postconditions

We assume that $\mathfrak{P} = \{\mu_1, \dots, \mu_n\}$ is a set of *parameters*. In FEHTs, we use assertions (formulas) over $\bigcup_{i=1}^{k+l} \mathcal{V}_i$, which we interpret as sets of (relational) states. A parametric assertion generalizes this by viewing an assertion as a function mapping *into* sets of (relational) states. Formally, a *parametric assertion* is a pair $(\Xi, \mathcal{C})$ where Ξ is a formula over $\bigcup_{i=1}^{k+l} \mathcal{V}_i \cup \mathfrak{P}$ (called the *function-formula*), and $\mathcal{C}$ is a formula over $\mathfrak{P}$ (called the *restriction-formula*).

Given a function-formula Ξ (over $\bigcup_{i=1}^{k+l} \mathcal{V}_i \cup \mathfrak{P}$) and a *parameter evaluation* $\kappa : \mathfrak{P} \to \mathbb{Z}$, we define $\Xi[\kappa]$ as the formula over $\bigcup_{i=1}^{k+l} \mathcal{V}_i$ where we fix concrete values for all parameters based on κ. We can thus view Ξ as a function mapping each parameter evaluation κ to the set of states encoded by $\Xi[\kappa]$. During our (forward style) analysis, we will use parameters to postpone nondeterministic choices in existentially quantified programs. Intuitively, for every parameter evaluation κ (i.e., any retrospective choice of the nondeterministic outcome), $\Xi[\kappa]$ should describe the reachable states (i.e., strongest postcondition) under those specific outcomes. However, not all concrete values for the parameters are valid in the sense that they correspond to nondeterministic outcomes that result in actual executions. To mitigate this, a parametric assertion $(\Xi, \mathcal{C})$ includes a restriction-formula $\mathcal{C}$ (over $\mathfrak{P}$) which *restrict the domain* of the function encoded by Ξ, i.e., we only consider those parameter evaluations that satisfy $\mathcal{C}$.

Example 2. Before proceeding with a formal development, let us discuss parametric assertions informally using an example. Let $\mathbb{P}_1 := x = \star \fatsemi \texttt{assume}(x \geq 9)$ and $\mathbb{P}_2 := y = \star \fatsemi \texttt{assume}(y \geq 2)$ and assume we want to prove the FEHT $\langle \top \rangle \mathbb{P}_1 \sim \mathbb{P}_2 \langle x = y \rangle$. To verify this tuple in a principled way, we are interested in potential postconditions Ψ, i.e., assertions Ψ such that $\langle \top \rangle \mathbb{P}_1 \sim \mathbb{P}_2 \langle \Psi \rangle$ is valid. For example, both $\Psi_1 = x \geq 9 \wedge y = 2$ and $\Psi_2 = x \geq 9 \wedge y = 3$ are valid postconditions, but – as already seen before – there does not exist a strongest assertion. Instead, we capture *multiple* postconditions using the parametric assertion $(\Xi, \mathcal{C})$ where $\Xi := x \geq 9 \wedge y = \mu$ and $\mathcal{C} := \mu \geq 2$ for some fresh parameter $\mu \in \mathfrak{P}$; we say $(\Xi, \mathcal{C})$ is a *parametric postcondition* for $(\top, \mathbb{P}_1, \mathbb{P}_2)$ (cf. Definition 2). Intuitively, we have used the parameter μ instead of assigning some fixed integer to y. For every concrete parameter evaluation $\kappa : \{\mu\} \to \mathbb{Z}$ such that $\kappa \models \mathcal{C}$,

formula $\Xi[\kappa]$ defines the reachable states when using $\kappa(\mu)$ for the choice of y. Observe how formula $\mathcal{C} = \mu \geq 2$ restricts the possible set of parameter values, i.e., we may only choose a value for y such that $\texttt{assume}(y \geq 2)$ holds. △

Definition 2. *A* parametric postcondition for $(\Phi, \mathbb{P}_1, \dots, \mathbb{P}_{k+l})$ *is a parametric assertion* $(\Xi, \mathcal{C})$ *with the following conditions. For all states* $\sigma_1, \dots, \sigma_{k+l}$*, and* $\sigma'_1, \dots, \sigma'_k$ *such that* $\bigoplus_{i=1}^{k+l} \sigma_i \models \Phi$ *and* $[\![\mathbb{P}_i]\!](\sigma_i, \sigma'_i)$ *for all* $i \in [1, k]$ *and any parameter evaluation* κ *such that* $\kappa \models \mathcal{C}$ *the following holds:* ***(1)*** *There exist states* $\sigma'_{k+1}, \dots, \sigma'_{k+l}$ *such that* $\bigoplus_{i=1}^{k+l} \sigma'_i \models \Xi[\kappa]$*, and* ***(2)*** *For every* $\sigma'_{k+1}, \dots, \sigma'_{k+l}$ *such that* $\bigoplus_{i=1}^{k+l} \sigma'_i \models \Xi[\kappa]$ *we have* $[\![\mathbb{P}_i]\!](\sigma_i, \sigma'_i)$ *for all* $i \in [k+1, k+l]$.

Condition **(1)** captures that no parameter evaluation may restrict universally quantified executions, i.e., if we fix any parameter evaluation κ and reachable final states for the universally quantified programs, $\Xi[\kappa]$ remains satisfiable. This effectively states that $\Xi[\kappa]$ *over-approximates* the set of executions of universally quantified programs. Condition **(2)** requires that all executions of existentially quantified programs allowed under a particular parameter evaluation are also valid executions, i.e., for any fixed parameter evaluation κ, $\Xi[\kappa]$ *under-approximates* the set of executions of the existentially quantified programs.

We can use parametric postconditions to prove FEHTs:

Theorem 3. *Let* $(\Xi, \mathcal{C})$ *be a parametric postcondition for* $(\Phi, \mathbb{P}_1, \dots, \mathbb{P}_{k+l})$*. If*

$$\forall_{x \in \mathcal{V}_1 \cup \dots \cup \mathcal{V}_k} x. \exists_{\mu \in \mathfrak{P}} \mu.\ \ \mathcal{C} \ \ \wedge \ \ \forall_{x \in \mathcal{V}_{k+1} \cup \dots \cup \mathcal{V}_{k+l}} x. (\Xi \Rightarrow \Psi)$$

holds, then the FEHT $\langle \Phi \rangle \mathbb{P}_1 \circledast \dots \circledast \mathbb{P}_k \sim \mathbb{P}_{k+1} \circledast \dots \circledast \mathbb{P}_{k+l} \langle \Psi \rangle$ *is valid.*

Here, we universally quantify over final states in $\mathbb{P}_1, \dots, \mathbb{P}_k$ and existentially quantify over parameter evaluations that satisfy $\mathcal{C}$ (recall that $\mathcal{C}$ only refers to $\mathfrak{P}$). The choice of the parameters can thus depend on the final states of universally quantified programs (as in the semantics of FEHTs). Afterward, we quantify (again universally) over final states of $\mathbb{P}_{k+1}, \dots, \mathbb{P}_{k+l}$ and state that if Ξ holds, so does the postcondition Ψ.

Example 3. Consider the FEHT and parametric postcondition from Example 2. Following Theorem 3, we construct the SMT formula $\forall x. \exists \mu. \mu \geq 2 \wedge \forall y. \big((x \geq 9 \wedge y = \mu) \Rightarrow x = y\big)$. This formula holds; the FEHT is valid. △

Note that $(\Xi, \bot)$ is always a parametric postcondition: no parameter evaluation satisfies $\bot$, so the conditions in Definition 2 are vacuously satisfied. However, $(\Xi, \bot)$ is useless when it comes to proving FEHTs via Theorem 3.

4.2 Generating Parametric Postconditions

Algorithm 1 computes a parametric postcondition based on the proof rules of FEHL from Section 3. As input, Algorithm 1 expects a formula Φ over $\bigcup_{i=1}^{k+l} \mathcal{V}_i \cup \mathfrak{P}$ – think of Φ as a precondition already containing some parameters – and two program lists $\overline{\chi_\forall}$ and $\overline{\chi_\exists}$. It outputs a parametric postcondition.

Algorithm 1 Parametric postcondition generation for FEHT verification

```
1  def genpp(Φ, χ∀, χ∃):
2   if χ∀ = χ∃ = ϵ:
3    return (Φ, ⊤) //(Done)
4   else if ∀ℙ ∈ χ∀ ∪ χ∃. ℙ = while(_,_)⨟_:
5    return genppLoops(Φ, χ∀, χ∃)
6   else if ∃ℙ ∈ χ∀. ℙ ≠ while(_,_)⨟_:
7    // Take a step in χ∀
8    match χ∀:
9    | skip ⊛ χ∀′: //(∀-Skip-E)
10     return genpp(Φ, χ∀′, χ∃)
11   | skip⨟ℙ ⊛ χ∀′: //(∀-Step)
12     return genpp(Φ, ℙ ⊛ χ∀′, χ∃)
13   | (ℙ1⨟ℙ2)⨟ℙ3 ⊛ χ∀′:
14     return genpp(Φ, ℙ1⨟(ℙ2⨟ℙ3) ⊛ χ∀′, χ∃)
15   | ℙ ⊛ χ∀′ when ℙ ≠ _⨟_: //(∀-Skip-I)
16     return genpp(Φ, ℙ⨟skip ⊛ χ∀′, χ∃)
17   | x = e⨟ℙ ⊛ χ∀′: //(∀-Step)
18     Φ′ := ∃x′.Φ[x′/x] ∧ x = e[x′/x]
19     return genpp (Φ′, ℙ ⊛ χ∀′, χ∃)
20   | if(b, ℙ1, ℙ2)⨟ℙ3 ⊛ χ∀′:
21     //(∀-If)
22     (Ξ1, C1) :=
23       genpp(Φ ∧ b, ℙ1⨟ℙ3 ⊛ χ∀′, χ∃)
24     (Ξ2, C2) :=
25       genpp(Φ ∧ ¬b, ℙ2⨟ℙ3 ⊛ χ∀′, χ∃)
26     return (Ξ1 ∨ Ξ2, C1 ∧ C2)
27   | assume(b)⨟ℙ ⊛ χ∀′: //(∀-Assume)
28     return genpp(Φ ∧ b, ℙ ⊛ χ∀′, χ∃)
29   | x = ⋆⨟ℙ ⊛ χ∀′: //(∀-Choice)
30     Φ′ := ∃x. Φ
31     return genpp(Φ′, ℙ ⊛ χ∀′, χ∃)
32   | ℙ ⊛ χ∀′: //(∀-Reorder)
33     return genpp(Φ, χ∀′ ⊛ ℙ, χ∃)
34  else:
35   // Take a step in χ∃
36   match χ∃:
37   | skip ⊛ χ∃′ | skip⨟ℙ ⊛ χ∃′
38   | (ℙ1⨟ℙ2)⨟ℙ3 ⊛ χ∃′
39   | ℙ ⊛ χ∃′ when ℙ ≠ _⨟_
40   | x = e⨟ℙ ⊛ χ∃′
41   | if(b, ℙ1, ℙ2)⨟ℙ3 ⊛ χ∃′:
42     //As in lines 9, 11, 17
43     //20, 13, and 15
44   | assume(b)⨟ℙ ⊛ χ∃′:
45     //(∃-Assume)
46     C_assume :=
47       ∀x∈V1∪⋯∪Vk+l. x. (Φ ⇒ b)
48     (Ξ, C) :=
49       genpp(Φ ∧ b, χ∀, ℙ ⊛ χ∃′)
50     return (Ξ, C ∧ C_assume)
51   | x = ⋆⨟ℙ ⊛ χ∃′: //(∃-Choice)
52     μ := freshParameter()
53     Φ′ := (∃x.Φ) ∧ x = μ
54     return genpp(Φ′, χ∀, ℙ ⊛ χ∃′)
55   | ℙ ⊛ χ∃′:
56     return genpp(Φ, χ∀, χ∃′ ⊛ ℙ)
```

Remark 2. For intuition, it is oftentimes helpful to consider Φ as a parameter-free formula over $\bigcup_{i=1}^{k+l} \mathcal{V}_i$. In this case, most of our steps correspond to the computation of the strongest postcondition [27,56,19] in a purely universal (k-safety) setting. △

Our algorithm analyses the structure of each program and applies the insights from FEHL: If $\overline{\chi_\forall}$ and $\overline{\chi_\exists}$ are empty, we return $(\Phi, \top)$ (line 3), i.e., we do not place any restrictions on the parameters. In case all programs are loops (line 5), we invoke a subroutine `genppLoops` (discussed in Section 4.3). Otherwise, some program has a non-loop statement at the top level, allowing further symbolic analysis. We consider possible steps in $\overline{\chi_\forall}$ (lines 7-33) and in $\overline{\chi_\exists}$ (lines 35-56).

We first consider the case where a universally quantified program has a non-loop statement at its top level (lines 7-33). In lines 9, 11, 13, and 15, we bring the first program into the form $\mathbb{P}_1 ⨟ \mathbb{P}_2$ where $\mathbb{P}_1 \neq \_ ⨟ \_$ by potentially inserting `skip` statements in line 15. For a program $x = e ⨟ \mathbb{P}$ (line 17), we use `(∀-Step)` to handle the assignment. Here, we can compute the strongest postcondition of the assignment as $\exists x'.\Phi[x'/x] \wedge x = e[x'/x]$ (using Floyd's forward running rule [35]). For conditionals (line 20), we analyze both branches under the strengthened precondition. As our analysis operates on parametric assertions, some of the parameters found in the precondition Φ can be restricted in *both branches*. After

we have computed a parametric postcondition for each branch, we therefore combine them into a parametric postcondition for the entire program by constructing the disjunction of the function-formulas Ξ_1 and Ξ_2 (describing the set of states reachable in either of the branches), and conjoining the restriction-formulas $\mathcal{C}_1$ and $\mathcal{C}_2$. For assume statements (line 27), we strengthen the precondition. For nondeterministic assignments $x = \star$ (line 29), we invalidate all knowledge about x. If a program matches none of the previous cases (line 33), it must be of the form $\texttt{while}(\_, \_)\,\texttt{;}\,\_$, and we move it to the end of $\overline{\chi_\forall}$, continuing the analysis of the renaming programs in the next recursive iteration. If no universally quantified program can be analyzed further, we continue the investigation with existentially quantified ones (lines 35-56). Many cases are analogous to the treatment in universally quantified programs (lines 37-43), but some cases are handled fundamentally differently: If we encounter an assume statement $\texttt{assume}(b)$ (line 45), we need to certify that b holds in all states in Φ (cf. **(∃-Assume)**). As we already hinted in Example 2, we accomplish this by restricting the viable set of parameters in Φ, i.e., we restrict the domain of the function formula Φ. Concretely, we consider the formula $\mathcal{C}_{assume} := \forall_{x \in \mathcal{V}_1 \cup \dots \cup \mathcal{V}_{k+l}}\, x.\, (\Phi \Rightarrow b)$ (which is a formula over $\mathfrak{P}$) that characterizes exactly those parameters that ensure that all states in Φ satisfy b. After analyzing the remaining programs, we then conjoin $\mathcal{C}_{assume}$ with the remaining restrictions.

Remark 3. As in Remark 2, we can consider the case where Φ contains no parameter. In this case, $\mathcal{C}_{assume}$ is a variable-free formula that is equivalent to $\top$ iff all states in Φ satisfy b. If Φ does *not* imply b (so $\mathcal{C}_{assume} \equiv \bot$), the resulting parametric postcondition thus cannot prove any FEHT via Theorem 3. △

For nondeterministic assignments $x = \star$ (line 51), we create a fresh parameter μ and continue the analysis under the precondition that $x = \mu$, effectively postponing the choice of a concrete value for x (cf. Example 2).

Example 4. Our algorithm will automatically compute the parametric postcondition from Example 2. In particular, for the $\texttt{assume}(y \geq 2)$ statement, we match line 45 with $\Phi = x \geq 9 \wedge y = \mu$ for $\mu \in \mathfrak{P}$ and compute $\mathcal{C}_{assume} := \forall x, y.\, \Phi \Rightarrow y \geq 2$, which is logically equivalent to $\mu \geq 2$. △

4.3 Generating Parametric Postconditions for Loops

We sketch the postcondition generation for loops in Algorithm 2. As input, `genppLoops` expects a precondition Φ over $\bigcup_{i=1}^{k+l} \mathcal{V}_i \cup \mathfrak{P}$ and universally and existentially quantified loop programs. In the first step, we guess a loop invariant $\mathbb{I}$ and counter values $c_1, \dots, c_{k+l} \in [1, B]$ (cf. **(Loop-Counting)**). In lines 4 and 5, we ensure that $\mathbb{I}$ is initial and guarantees simultaneous termination by computing restrictions $\mathcal{C}_{init}$ and $\mathcal{C}_{sim}$ on the parameters present in Φ (similar to `assume` statements in line 45 of Algorithm 1). Again, in the special case where Φ contains no parameter (as is, e.g., the case when applying our algorithm to k-safety properties), $\mathcal{C}_{init}$ (resp. $\mathcal{C}_{sim}$) is equivalent to $\top$ iff the invariant is initial

Algorithm 2 Parametric postcondition generation for loops

1 **def** genppLoops(Φ, $\circledast_{i=1}^{k}$ (while$(b_i, \mathbb{P}_i)$; $\mathbb{Q}_i$), $\circledast_{i=k+1}^{k+l}$ (while$(b_i, \mathbb{P}_i)$; $\mathbb{Q}_i$)):
2 $\quad \mathbb{I}, c_1, \ldots, c_{k+l}$:= guessInvariantAndCounts()
3 $\quad B$:= max$(c_1, \ldots, c_{k+l})$
4 $\quad \mathcal{C}_{init}$:= $\forall_{x \in \mathcal{V}_1 \cup \cdots \cup \mathcal{V}_{k+l}}\, x.\, (\Phi \Rightarrow \mathbb{I})$
5 $\quad \mathcal{C}_{sim}$:= $\forall_{x \in \mathcal{V}_1 \cup \cdots \cup \mathcal{V}_{k+l}}\, x.\, (\mathbb{I} \Rightarrow \bigwedge_{i=2}^{k+l} b_1 \leftrightarrow b_i)$
6 $\quad \Xi_1$:= $\mathbb{I}$
7 $\quad$ **for** j **from** 1 **to** B:
8 $\quad\quad (\Xi_{j+1}, \mathcal{C}_{j+1})$:= genpp$(\Xi_j \wedge \bigwedge_{i=1|c_i \geq j}^{k+l} b_i, \circledast_{i=1|c_i \geq j}^{k} \mathbb{P}_i, \circledast_{i=k+1|c_i \geq j}^{k+l} \mathbb{P}_i)$
9 $\quad\quad \mathcal{C}_{j+1}^{cont}$:= $\forall_{x \in \mathcal{V}_1 \cup \cdots \cup \mathcal{V}_{k+l}}\, x.\, (\Xi_{j+1} \Rightarrow \bigwedge_{i=1|c_i > j}^{k+l} b_i)$
10 $\quad \mathcal{C}_{ind}$:= $\forall_{x \in \mathcal{V}_1 \cup \cdots \cup \mathcal{V}_{k+l}}\, x.\, (\Xi_{B+1} \Rightarrow \mathbb{I})$
11 $\quad (\Xi_{rem}, \mathcal{C}_{rem})$:= genpp$(\mathbb{I} \wedge \bigwedge_{i=1}^{k+l} \neg b_i, \circledast_{i=1}^{k} \mathbb{Q}_i, \circledast_{i=k+1}^{k+l} \mathbb{Q}_i)$
12 $\quad$ **return** $(\Xi_{rem}, \mathcal{C}_{init} \wedge \mathcal{C}_{sim} \wedge \bigwedge_{j=2}^{B+1} \mathcal{C}_j \wedge \bigwedge_{j=2}^{B+1} \mathcal{C}_j^{cont} \wedge \mathcal{C}_{ind} \wedge \mathcal{C}_{rem})$

(resp. guarantees simultaneous termination). Afterward, we check the validity of the guessed counter values $c_1, \ldots, c_{k+l}$. For each j from 1 to B, we compute a parametric postcondition $(\Xi_{j+1}, \mathcal{C}_{j+1})$ for the bodies of all loops that should be executed at least j times (i.e., $c_i \geq j$) starting from precondition Ξ_j via a (mutually recursive) call to genpp (line 8). To ensure valid derivation using **(Loop-Counting)** we need to ensure that – in Ξ_{j+1} – the guard of all loops that we want to execute *more* than j times still evaluates to true. We ensure this by computing the restriction-formula $\mathcal{C}_{j+1}^{cont}$, which restricts the parameters (both those already present in the precondition Φ and those added during the analysis of the loop bodies) such that all states in Ξ_{j+1} fulfill the guards of all loops with $c_i > j$ (line 9). After we have symbolically executed all loops the desired number of times, we construct a parameter restriction $\mathcal{C}_{ind}$ that ensures that we end within the invariant, i.e., $\Xi_{B+1} \Rightarrow \mathbb{I}$ (line 10). In the last step, we compute a parametric postcondition $(\Xi_{rem}, \mathcal{C}_{rem})$ for the program suffix executed after the loops. We return the parametric postcondition that consists of the function-formula Ξ_{rem} and the conjunction of all restriction-formulas.

4.4 The Main Verification

From the soundness of FEHL (Theorem 1) we directly get:

Proposition 1. *genpp$(\Phi, \overline{\chi_\forall}, \overline{\chi_\exists})$ computes some parametric postcondition for* $(\Phi, \overline{\chi_\forall}, \overline{\chi_\exists})$.

Given an FEHT $\langle \Phi \rangle \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle$, we can thus invoke genpp$(\Phi, \overline{\chi_\forall}, \overline{\chi_\exists})$ to compute a parametric postcondition, which (if strong enough) allows us to prove that $\langle \Phi \rangle \overline{\chi_\forall} \sim \overline{\chi_\exists} \langle \Psi \rangle$ is valid via Theorem 3. If the postcondition is too weak, we can re-run genpp using updated invariant guesses (cf. Section 5). For loop-free programs, it is easy to see that genpp computes the "strongest possible" parametric postcondition (it effectively executes the programs symbolically without incurring the imprecision inserted by loop invariants). In this case, the query from

Theorem 3 holds if and only if the FEHT is valid; our algorithm thus constitutes a *complete* verification method.

Invalid FEHTs. We stress that the goal of our algorithm is the verification of FEHTs and not proving that an FEHT is *invalid*. For k-safety properties, a refutation (counterexample) consists of a k-tuple of concrete executions that violate the property [56,19]. In contrast, refuting an FEHT corresponds to *proving* a $\exists^*\forall^*$ property, an orthogonal problem that requires independent proof ideas.

5 Implementation and Experiments

We have implemented our verification algorithm in a tool called `ForEx` [9] (short for **For**all **Ex**ists Verification), supporting programs in a minimalistic `C`-like language that features basic control structures (cf. Section 2), arrays, and bitvectors. `ForEx` uses `Z3` [48] to discharge SMT queries and supports the theory of linear integer arithmetic, the theory of arrays, and the theory of finite bitvectors. Compared to the presentation in Section 4, we check satisfiability of restriction-formulas *eagerly*: For example, in Algorithm 2, we compute multiple restriction-formulas and return their conjunction. In `ForEx`, we immediately check these intermediate restrictions for satisfiability; if any restriction is unsatisfiable on its own, any conjunction involving it will be as well, so we can abort the analysis early and re-start parts of the analysis using, e.g., updated invariants and counter values.

5.1 Loop Invariant Generation

Our loop invariant generation and counter value inference follows a standard guess-and-check procedure [34,54,56,19,53], i.e., we generate promising candidates by combining expressions found in the programs and equalities between variables in the loop guards. In most loops, there exist "anchor" variables that effectively couple executions of multiple loops together [56,19]; even in asynchronous cases like Example 1. Exploring more advanced invariant generation techniques is interesting future work. However – even in the simpler setting of k-safety properties – many tools currently rely on a guess-and-check approach [56,19]. We maintain a lattice of possible candidates ordered by implication, which allows us for efficient pruning. For example, if the current candidate is not initial (i.e., $\mathcal{C}_{init}$ computed in line 4 of Algorithm 2 is unsatisfiable), we do not need to consider stronger candidates. Likewise, if the candidate does not ensure simultaneous termination ($\mathcal{C}_{sim}$) we can prune all weaker invariants.

5.2 Experiments

We evaluate `ForEx` in various settings where FEHT-like specifications arise. We compare with `HyPA` (a predicate-abstraction-based solver) [12], `PCSat` (a constraint-based solver that relies on predicate templates) [57], and `HyPro` (a model-checker for $\forall^*\exists^*$ properties in *finite-state* systems) [11]. Our results were obtained on a M1 Pro CPU with 32GB of memory.

Instance	$t_{\texttt{HyPA}}$	$t_{\texttt{ForEx}}$
DoubleSquareNI†	67.12	**0.71**
Exp1x3	3.79	**0.30**
Fig3	8.78	**0.39**
DoubleSquareNIff	4.91	**0.37**
Fig2†	17.7	**0.73**
ColItemSymm	15.51	**0.20**
CounterDet	5.28	**0.55**
MultEquiv	13.13	**0.60**
HalfSquareNI	**68.04**	-
SquaresSum	**17.03**	-
ArrayInsert	**16.17**	-

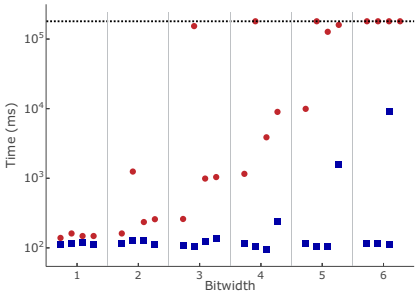
(a)

Instance	$t_{\texttt{HyPA}}$	$t_{\texttt{ForEx}}$
NonDetAdd	3.63	**0.76**
CounterSum	5.05	**1.95**
AsynchGNI	5.20	**0.69**
CompilerOpt1	1.79	**0.59**
CompilerOpt2	2.71	**1.02**
Refine	10.1	**0.57**
Refine2	9.87	**0.64**
Smaller	2.21	**0.69**
CounterDiff	8.05	**0.63**
Fig. 3	8.92	**0.57**

(b)

Instance	$t_{\texttt{PCSat}}$	$t_{\texttt{ForEx}}$
TI_GNI_hFF	26.2	**0.58**
TI_GNI_hTT	32.5	**0.10**
TI_GNI_hFT†,‡	36.2	**0.70**
TS_GNI_hFF	36.6	**0.58**
TS_GNI_hTT‡	96.2	**0.16**
TS_GNI_hFT†,‡	123.3	**2.88**
TI_GNI_hTF	**26.1**	-
TS_GNI_hTF	**44.1**	-

(c)



(d)

Fig. 5: In Tables 5a and 5b, we compare `ForEx` with `HyPA` [12] on k-safety and $\forall^*\exists^*$ properties, respectively. For instances marked with †, `ForEx` required additional user-provided invariant hints. In Table 5c, we compare `ForEx` with `PCSat` [57]. For instances marked with ‡, `PCSat` required additional invariant hints. In Figure 5d, we compare the running time of `ForEx` (■) and `HyPro` [11] (●). We check each of the 4 GNI instances from [11] with varying bitwidth. The timeout is set to 3 min (marked by the horizontal dotted line).

Limitations of `ForEx`'s Loop Alignment. Before we evaluate `ForEx` on $\forall^*\exists^*$ properties, we investigate the counting-based loop alignment principle underlying `ForEx`. We collect the k-safety benchmarks from `HyPA` [12] (which themself were collected from multiple sources [32,31,55,57]) and depict the verification results in Table 5a. We observe that `ForEx` can verify many of these instances. As it explores a restricted class of loop alignments (guided by `(Loop-Counting)`), it is more efficient on the instances it can solve. However, for some of the instances, `ForEx`'s counting-based alignment is insufficient. Instead, these in-

stances require a loop alignment that is context-dependent, i.e., the alignment is chosen based on the current state of the programs [12,55,32,57].

***ForEx** and **HyPA**.* `HyPA` [12] explores a liberal program alignment by exploring a user-provided predicate abstraction. The verification instances considered in [12] include a range of $\forall^*\exists^*$ properties on very small programs, including, e.g., GNI and refinement properties. In Table 5b, we compare the running time of `ForEx` with that of `HyPA` (using the user-defined predicates for its abstraction).[1] We observe that `ForEx` can verify the instances significantly quicker. Moreover, we stress that `ForEx` solves a much more challenging problem as it analyzes the program *fully automatically* without any user intervention.

***ForEx** and **PCSat**.* Unno et al. [57] present an extension of constraint Horn clauses, called pfwCSP, that is able to express a range of relational properties (including $\forall^*\exists^*$ properties). Their custom pfwCSP solver (called `PCSat`) instantiates predicates with user-provided templates. We compare `PCSat` and `ForEx` in Table 5c. `ForEx` can verify 6 out of the 8 $\forall^*\exists^*$ instances. `ForEx` currently does not support termination proofs for loops in existentially quantified programs (which are needed for TI_GNI_HTF and TS_GNI_HTF), whereas `PCSat` features loop variant templates and can thus reason about the termination of existentially quantified loops in isolation. In the instances that `ForEx` can solve, it is much faster. We conjecture that this is due to the fact that the constraints generated by `ForEx` can be solved directly by SMT solvers, whereas `PCSat`'s pfwCSP constraints first require a custom template instantiation.

***ForEx** and **HyPro**.* Programs whose variables have a finite domain (e.g., boolean) can be checked using explicit-state techniques developed for logics such as HyperLTL [20]. We verify GNI on variants of the four boolean programs from [11] with a varying number of bits. We compare `ForEx` with the HyperLTL verifier `HyPro` [11], which converts a program into an explicit-state transition system. We depict the results in Figure 5d. We observe that, with increasing bitwidth, the running time of explicit-state model-checking increases exponentially (note that the scale is logarithmic). In contrast, `ForEx` can employ symbolic bitvector reasoning, resulting in orders of magnitude faster verification.

6 Related Work

Most methods for k-safety verification are centered around the self-composition of a program [6] and often improve upon a naïve self-composition by, e.g., exploiting the commutativity of statements [55,31,32,29]. Relational program logics

[1] The properties checked by `HyPA` [12] are temporal, i.e., properties about the infinite execution of programs of the form `while`$(\top, \mathbb{P})$. To make such programs analyzable in `ForEx` (which reasons about finite executions), we replaced the *infinite* loop with a loop that executes $\mathbb{P}$ some fixed (but arbitrary) number of times.

for k-safety offer a rich set of rules to *over*-approximate the program behavior [7,60,56,49,28,3,8]. Recently, much effort has been made to employ under-approximate methods that find bugs instead of proving their absence; so far, mostly for unary (non-hyper) properties [50,58,52,47,42,17,62,24].

Dardinier et al. [25] propose *Hyper Hoare Logic* – a logic that can express *arbitrary* hyperproperties, but requires manual deductive reasoning. Dickerson et al. [26] introduce RHLE, a program logic for the verification of $\forall^*\exists^*$ properties, focusing on the composition (and under-approximation) of function calls. They present a weakest-precondition-based verification algorithm that aligns loops in lock-step via user-provided loop invariants. Unno et al. [57] present an extension of constraint Horn-clauses (called pfwCSP). They show that pfwCSP can encode many relational verification conditions, including many hyperliveness properties like GNI (see Section 5). Compared to the pfwCSP encoding, we explore a less liberal program alignment (guided by `(Loop-Counting)`). However, we gain the important advantage of generating standard (first-order) SMT constraints that can be handled using existing SMT solvers (which shows significant performance improvement, cf. Section 5).

Most work on the verification of hyperliveness has focused on more general *temporal* properties, i.e., properties that reason about infinite executions, based on logics such as HyperLTL [20,33,13]. Coenen et al. [22] study a method for verifying hyperliveness in *finite*-state transition systems using strategies to resolve existential quantification. This approach is also applicable to infinite-state systems by means of an abstraction [12,39] (see `HyPA` in Section 5). Bounded model-checking (BMC) for hyperproperties [38] unrolls the system to a fixed bound and can, e.g., find violations to GNI. Existing BMC tools target finite-state (boolean) systems and construct QBF formulas; lifting this to support infinite-state systems by constructing SMT constraints is an interesting future work and could, e.g., complement `ForEx` in the refutation of FEHTs.

7 Conclusion

We have studied the automated program verification of relational $\forall^*\exists^*$ properties. We developed a constraint-based verification algorithm that is rooted in a sound-and-complete program logic and uses a (parametric) postcondition computation. Our experiments show that – while our logic-guided tool explores a restricted class of possible loop alignments – it succeeds in many of the instances we tested. Moreover, the use of off-the-shelf SMT solvers results in faster verification, paving the way toward a future of fully automated tools that can check important hyperliveness properties such as GNI and opacity.

Acknowledgments. This work was supported by the European Research Council (ERC) Grant HYPER (101055412), and by the German Research Foundation (DFG) as part of TRR 248 (389792660).

Data Availability Statement. `ForEx` is available at [9].

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Softw. Syst. Model. (2005). https://doi.org/10.1007/s10270-004-0058-x
2. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. (1985). https://doi.org/10.1016/0020-0190(85)90056-0
3. Antonopoulos, T., Koskinen, E., Le, T.C., Nagasamudram, R., Naumann, D.A., Ngo, M.: An algebra of alignment for relational verification. Proc. ACM Program. Lang. (POPL) (2023). https://doi.org/10.1145/3571213
4. Assaf, M., Naumann, D.A., Signoles, J., Totel, E., Tronel, F.: Hypercollecting semantics and its application to static analysis of information flow. In: Symposium on Principles of Programming Languages, POPL 2017 (2017). https://doi.org/10.1145/3009837.3009889
5. Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: Asymmetric product programs for relational program verification. In: International Symposium on Logical Foundations of Computer Science, LFCS 2013 (2013). https://doi.org/10.1007/978-3-642-35722-0_3
6. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Math. Struct. Comput. Sci. (2011). https://doi.org/10.1017/S0960129511000193
7. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Symposium on Principles of Programming Languages, POPL 2004 (2004). https://doi.org/10.1145/964001.964003
8. Beringer, L.: Relational decomposition. In: International Conference on Interactive Theorem Proving, ITP 2011 (2011). https://doi.org/10.1007/978-3-642-22863-6_6
9. Beutner, R.: ForEx: Automated Software Verification of Hyperliveness (2023). https://doi.org/10.5281/zenodo.10436583
10. Beutner, R.: Automated software verification of hyperliveness. CoRR (2024)
11. Beutner, R., Finkbeiner, B.: Prophecy variables for hyperproperty verification. In: Computer Security Foundations Symposium, CSF 2022 (2022). https://doi.org/10.1109/CSF54842.2022.9919658
12. Beutner, R., Finkbeiner, B.: Software verification of hyperproperties beyond k-safety. In: International Conference on Computer Aided Verification, CAV 2022 (2022). https://doi.org/10.1007/978-3-031-13185-1_17
13. Beutner, R., Finkbeiner, B.: AutoHyper: Explicit-state model checking for HyperLTL. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023 (2023). https://doi.org/10.1007/978-3-031-30823-9_8
14. Beutner, R., Finkbeiner, B.: Model checking omega-regular hyperproperties with AutoHyperQ. In: International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2023 (2023). https://doi.org/10.29007/1XJT
15. Beutner, R., Finkbeiner, B., Frenkel, H., Metzger, N.: Second-order hyperproperties. In: International Conference on Computer Aided Verification, CAV 2023 (2023). https://doi.org/10.1007/978-3-031-37703-7_15
16. Biewer, S., Dimitrova, R., Fries, M., Gazda, M., Heinze, T., Hermanns, H., Mousavi, M.R.: Conformance relations and hyperproperties for doping detection in time and space. Log. Methods Comput. Sci. (2022). https://doi.org/10.46298/lmcs-18(1:14)2022
17. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: A correctness and incorrectness program logic. J. ACM (2023). https://doi.org/10.1145/3582267

18. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity and robustness of programs. Commun. ACM (2012). https://doi.org/10.1145/2240236.2240262
19. Chen, J., Feng, Y., Dillig, I.: Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In: Conference on Computer and Communications Security, CCS 2017 (2017). https://doi.org/10.1145/3133956.3134058
20. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: International Conference om Principles of Security and Trust, POST 2014 (2014). https://doi.org/10.1007/978-3-642-54792-8_15
21. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. (2010). https://doi.org/10.3233/JCS-2009-0393
22. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: International Conference on Computer Aided Verification, CAV 2019 (2019). https://doi.org/10.1007/978-3-030-25540-4_7
23. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM J. Comput. (1978). https://doi.org/10.1137/0207005
24. Cousot, P.: Calculational design of [in]correctness transformational program logics by abstract interpretation. Proc. ACM Program. Lang. (POPL) (2024)
25. Dardinier, T., Müller, P.: Hyper hoare logic: (dis-)proving program hyperproperties. CoRR (2023). https://doi.org/10.48550/arXiv.2301.10037
26. Dickerson, R., Ye, Q., Zhang, M.K., Delaware, B.: RHLE: modular deductive verification of relational ∀∃ properties. In: Asian Symposium on Programming Languages and Systems, APLAS 2022 (2022). https://doi.org/10.1007/978-3-031-21037-2_4
27. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Texts and Monographs in Computer Science, Springer (1990). https://doi.org/10.1007/978-1-4612-3228-5
28. D'Osualdo, E., Farzan, A., Dreyer, D.: Proving hypersafety compositionally. Proc. ACM Program. Lang. (OOPSLA) (2022). https://doi.org/10.1145/3563298
29. Eilers, M., Müller, P., Hitz, S.: Modular product programs. ACM Trans. Program. Lang. Syst. (2020). https://doi.org/10.1145/3324783
30. Farina, G.P., Chong, S., Gaboardi, M.: Relational symbolic execution. In: International Symposium on Principles and Practice of Programming Languages, PPDP 2019 (2019). https://doi.org/10.1145/3354166.3354175
31. Farzan, A., Vandikas, A.: Automated hypersafety verification. In: International Conference on Computer Aided Verification, CAV 2019 (2019). https://doi.org/10.1007/978-3-030-25540-4_11
32. Farzan, A., Vandikas, A.: Reductions for safety proofs. Proc. ACM Program. Lang. (POPL) (2020). https://doi.org/10.1145/3371081
33. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: International Conference on Computer Aided Verification, CAV 2015 (2015). https://doi.org/10.1007/978-3-319-21690-4_3
34. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: International Symposium of Formal Methods Europe, FME 2001 (2001). https://doi.org/10.1007/3-540-45251-6_29
35. Floyd, R.W.: Assigning meanings to programs. Program Verification: Fundamental Issues in Computer Science (1993)
36. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Symposium on Principles of Programming Languages, POPL 2004 (2004). https://doi.org/10.1145/964001.964021

37. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM (1969). https://doi.org/10.1145/363235.363259
38. Hsu, T., Sánchez, C., Bonakdarpour, B.: Bounded model checking for hyperproperties. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2021 (2021). https://doi.org/10.1007/978-3-030-72016-2_6
39. Itzhaky, S., Shoham, S., Vizel, Y.: Hyperproperty verification as CHC satisfiability. CoRR (2023). https://doi.org/10.48550/arXiv.2304.12588
40. King, J.C.: Symbolic execution and program testing. Commun. ACM (1976). https://doi.org/10.1145/360248.360252
41. Kovács, M., Seidl, H., Finkbeiner, B.: Relational abstract interpretation for the verification of 2-hypersafety properties. In: Conference on Computer and Communications Security, CCS 2013 (2013). https://doi.org/10.1145/2508859.2516721
42. Maksimovic, P., Cronjäger, C., Lööw, A., Sutherland, J., Gardner, P.: Exact separation logic: Towards bridging the gap between verification and bug-finding. In: European Conference on Object-Oriented Programming, ECOOP 2023 (2023). https://doi.org/10.4230/LIPICS.ECOOP.2023.19
43. Mastroeni, I., Pasqua, M.: Verifying bounded subset-closed hyperproperties. In: International Symposium on Static Analysis, SAS 2018 (2018). https://doi.org/10.1007/978-3-319-99725-4_17
44. Mastroeni, I., Pasqua, M.: Statically analyzing information flows: an abstract interpretation-based hyperanalysis for non-interference. In: Symposium on Applied Computing, SAC 2019 (2019). https://doi.org/10.1145/3297280.3297498
45. McCullough, D.: Noninterference and the composability of security properties. In: Symposium on Security and Privacy, SP 1988. IEEE Computer Society (1988). https://doi.org/10.1109/SECPRI.1988.8110
46. McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In: Symposium on Research in Security and Privacy, SP 1994 (1994). https://doi.org/10.1109/RISP.1994.296590
47. Möller, B., O'Hearn, P.W., Hoare, T.: On algebra of program correctness and incorrectness. In: International Conference on Relational and Algebraic Methods in Computer Science, RAMiCS 2021 (2021). https://doi.org/10.1007/978-3-030-88701-8_20
48. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
49. Nagasamudram, R., Naumann, D.A.: Alignment completeness for relational hoare logics. In: Symposium on Logic in Computer Science, LICS 2021 (2021). https://doi.org/10.1109/LICS52264.2021.9470690
50. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. (POPL) (2020). https://doi.org/10.1145/3371078
51. Pasareanu, C.S., Visser, W.: Verification of Java programs using symbolic execution and invariant generation. In: International Workshop on Model Checking Software, SPIN 2004 (2004). https://doi.org/10.1007/978-3-540-24732-6_13
52. Raad, A., Berdine, J., Dang, H., Dreyer, D., O'Hearn, P.W., Villard, J.: Local reasoning about the presence of bugs: Incorrectness separation logic. In: International Conference on Computer Aided Verification, CAV 2020 (2020). https://doi.org/10.1007/978-3-030-53291-8_14
53. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: International Conference on Computer Aided Verification, CAV 2014 (2014). https://doi.org/10.1007/978-3-319-08867-9_6

54. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: European Symposium on Programming Languages and Systems, ESOP 2013 (2013). https://doi.org/10.1007/978-3-642-37036-6_31
55. Shemer, R., Gurfinkel, A., Shoham, S., Vizel, Y.: Property directed self composition. In: International Conference on Computer Aided Verification, CAV 2019 (2019). https://doi.org/10.1007/978-3-030-25540-4_9
56. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: Conference on Programming Language Design and Implementation, PLDI 2016 (2016). https://doi.org/10.1145/2908080.2908092
57. Unno, H., Terauchi, T., Koskinen, E.: Constraint-based relational verification. In: International Conference on Computer Aided Verification, CAV 2021 (2021). https://doi.org/10.1007/978-3-030-81685-8_35
58. de Vries, E., Koutavas, V.: Reverse hoare logic. In: International Conference on Software Engineering and Formal Methods, SEFM 2011. LNCS (2011). https://doi.org/10.1007/978-3-642-24690-6_12
59. Wirth, N.: Program development by stepwise refinement. Commun. ACM (1971). https://doi.org/10.1145/362575.362577
60. Yang, H.: Relational separation logic. Theor. Comput. Sci. (2007). https://doi.org/10.1016/j.tcs.2006.12.036
61. Zhang, K., Yin, X., Zamani, M.: Opacity of nondeterministic transition systems: A (bi)simulation relation approach. IEEE Trans. Autom. Control. (2019). https://doi.org/10.1109/TAC.2019.2908726
62. Zilberstein, N., Dreyer, D., Silva, A.: Outcome logic: A unifying foundation for correctness and incorrectness reasoning. Proc. ACM Program. Lang. (OOPSLA) (2023). https://doi.org/10.1145/3586045