RHLE: Modular Deductive Verification of Relational ∀∃ Properties

Robert Dickerson(✉), Qianchuan Ye, Michael K. Zhang, and Benjamin Delaware

Purdue University, West Lafayette 47907, USA
{dicker18,ye202,bendy}@purdue.edu,
michael.k.zhang@alumni.purdue.edu

Abstract. Hoare-style program logics are a popular and effective technique for software verification. Relational program logics are an instance of this approach that enables reasoning about relationships between the execution of two or more programs. Existing relational program logics have focused on verifying that *all* runs of a collection of programs do not violate a specified relational behavior. Several important relational properties, including refinement and noninterference, do not fit into this category, as they also mandate the *existence* of specific desirable executions. This paper presents RHLE, a logic for verifying these sorts of relational ∀∃ properties. Key to our approach is a novel form of function specification that employs a variant of ghost variables to ensure that valid implementations exhibit certain behaviors. We have used a program verifier based on RHLE to verify a diverse set of relational ∀∃ properties drawn from the literature.

1 Introduction

Hoare-style program logics are a popular and effective verification technique. Starting with Hoare's seminal paper [20], this approach has been adapted to cover a variety of programming languages and assertions [3,21,28,32,34]. These logics typically feature several pleasant properties: they can be declaratively specified via a set of rules over the syntax of the target programming language, they permit compositional reasoning over individual program components, and they often admit effective automated verification procedures. Most of these logics focus on proving *safety* properties of *single* programs, i.e., that executing a program in a valid initial state never results in a state violating a postcondition.

Not all program behaviors fall into this category, however. As one example, consider the common scenario where a developer decides they want to migrate a hand-rolled implementation of a function to one that uses a third-party library. Figure 1 gives a concrete example of this situation. The program on the left, $\texttt{sample}_1$, uses a random number generator to directly sample a subset of an array. The program on the right, $\texttt{sample}_2$, opts to delegate the task to an external list library which supports shuffling and constructing sublists. While $\texttt{sample}_1$ works *with replacement* (the same elements may be sampled multiple times), $\texttt{sample}_2$ works *without*

I. Sergey (Ed.): APLAS 2022, LNCS 13658, pp. 67–87, 2022.
https://doi.org/10.1007/978-3-031-21037-2_4

```
int[] sample₁(int[] arr,
              int size) {
  assert(size <= arr.length);
  int[] samp = new int[size];
  for (i in [0..size]) {
    int j = randB(arr.length);
    samp[i] = arr[j];
  }
  return samp;
}
```

```
int[] sample₂(int[] arr,
              int size) {
  assert(size <= arr.length);
  list = new List(arr);
  perm = list.permute();
  samp = perm.sublist(size);
  return samp.toArray();
}
```

Fig. 1. An example migration of a function which randomly samples a list of integers with replacement to a function which samples without replacement. The original program ($\texttt{sample}_1$) uses a function which generates random numbers, while the migrated program ($\texttt{sample}_2$) uses a list abstraction with a `permute` operation.

replacement (an element may be sampled at most once). In order to ensure that this change does not break things, the developer may wish to verify that $\texttt{sample}_2$ does not do anything that $\texttt{sample}_1$ could not, i.e., that the updated function *refines* the original. Notably, this refinement property relates the behavior of *multiple* programs. In addition, it does not have the form of a standard safety property. The developer does not want to enforce that $\texttt{sample}_2$ produces *every* permutation that the hand-rolled implementation does; rather, they wish to ensure it does not start returning previously impossible samples.

```
int encode(int msgᴴ) {
  int keyᴴ = randB(MAX_INT);
  int encᴸ = msgᴴ xor keyᴴ;
  return (keyᴴ, encᴸ);
}
```

As another example, consider the `encode` function on the right which performs a simple xor cipher. This function takes a single high-security argument, $\texttt{msg}^{\texttt{H}}$, and returns a pair of high-security and low-security results, $\texttt{key}^{\texttt{H}}$ and $\texttt{enc}^{\texttt{L}}$, respectively. The function encodes its argument by first generating a random key (`randB` returns a random value between 0 and its argument), taking the `xor` of the key and the message, and finally returning the key along with the encoded message. The developer may wish to guarantee an attacker can learn nothing about the secret message given only the encoded message. Whether or not `encode` meets this *generalized noninterference* [26] property crucially depends on the behavior of `randB`: if the attacker knows this function *always* returns `3`, for example, they can decipher any encoded message. We can again frame this behavior as a relational property between the executions of two programs (in this case calls to `encode` with arbitrary arguments $\texttt{msg}_1^{\texttt{H}}$ and $\texttt{msg}_2^{\texttt{H}}$): every execution of $\texttt{encode(msg}_1^{\texttt{H}}\texttt{)}$ must have a corresponding execution of $\texttt{encode(msg}_2^{\texttt{H}}\texttt{)}$ that returns the same low-security encoded value.

In both examples, the desired behavior has the shape *for all* executions of some program, *there exists* a corresponding execution of a second program that is somehow related. Thus, we call these properties *relational* $\forall\exists$ properties. While several *relational program logics* have been developed for reasoning about the behavior of multiple programs [8,9,36], all have focused on relational *safety* properties, i.e.,

$$
\begin{array}{llll}
n \in \mathbb{N} \quad x, y \in \mathcal{V} & & s ::= \mathsf{skip} \mid s;\ s \\
f, g \in \mathcal{N} \quad \sigma \in \mathcal{V} \to \mathbb{N} & & \quad \mid \mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ s \\
a ::= n \mid x \mid a + a \mid a - a \mid a * a & & \quad \mid \mathsf{while}\ b\ \mathsf{do}\ s\ \mathsf{end} \\
b ::= \mathsf{true} \mid \mathsf{false} & & \quad \mid x := a \mid x := \mathsf{havoc} \mid x := f(\overline{a}) \\
\quad \mid a = a \mid a < a \mid \neg b \mid b \wedge b & & FD ::= \ \mathbf{def}\ f(\overline{x})\ \{s; \mathbf{return}\ a\}
\end{array}
$$

Fig. 2. Syntax of FunIMP.

that *all* the final states of multiple programs satisfy some relational postcondition. Unfortunately, in the presence of nondeterminism, none of these logics are capable of verifying relational ∀∃ properties such as refinement and generalized noninterference. The need to reason about nondeterminism naturally arises in the presence of external functions like `permute` in Fig. 1, where specifications are used to approximate the behavior of multiple possible implementations.

This paper addresses this gap by introducing RHLE, a relational program logic for reasoning about ∀∃ properties. Key to our approach is a novel form of function specifications which approximate the set of behaviors a valid implementation *must* exhibit. These specifications use a novel variant of ghost variables, which we call *choice variables*, that guarantee the existence of required behaviors. RHLE admits a modular reasoning principle, where any properties verified against a set of function specifications continue to hold whenever the program is linked to any satisfying implementation. While techniques based on Constrained Horn Clauses [38] and model checking [25] have recently been developed that are capable of reasoning about ∀∃ properties, RHLE is, to the best of our knowledge, the first Hoare-style program logic for doing so. We have used a verifier based on RHLE to verify a range of ∀∃ properties including refinement, noninterference (with and without delimited release), semantic parameter usage, and flaky tests.

We begin by defining a core imperative language with function calls (Sect. 2) equipped with semantics for both over- and under-approximating function behaviors (Sect. 3). We next present RHLE, and a corresponding verification algorithm for verifying ∀∃ properties (Sect. 5). We evaluate our approach by applying an implementation of this algorithm to verify a diverse set of relational properties (Sect. 6). We conclude with an examination of related work (Sect. 7). We have formalized the details of our approach in the Coq proof assistant; this development is available in the supplementary materials of this paper [16]. Our verification tool and benchmark suite are also publicly available [16,17].

2 The FunIMP Language

We begin with the definition of FunIMP, a core imperative language with function calls $x := f(\overline{a})$ and nondeterministic variable assignment $x := \mathsf{havoc}$. The full syntax of FunIMP is presented in Fig. 2. The calculus is parameterized over disjoint sets of identifiers for program variables $\mathcal{V}$ and function names $\mathcal{N}$.

Functions have a fixed arity. Function definitions consist of a sequence of statements followed by an expression that computes the result of the function. For brevity, we denote sequences $x_1, \ldots, x_n$ as $\overline{x}$. For ease of presentation, we treat functions as returning a single value, although it is straightforward to extend FunIMP to allow for multiple return values: $(x, y, \ldots) := f(\overline{a})$. Our verification tool, ORHLE (see Sect. 6), uses such an extension to model functions which mutate their arguments.

The semantics of FunIMP programs are defined via a standard big-step evaluation relation from initial to final program states. States are mappings from variables to integers, and are usually notated as σ. We write $[x \mapsto a]\sigma$ to refer to state σ updated with a mapping from x to a. The evaluation rules are parameterized over an *implementation context*, a mapping $I \in \mathcal{N} \rightarrow FD$ from function names to their definitions, which is used to evaluate function calls:

$$\frac{\begin{array}{c} I(f) = \textbf{def } f(\overline{x})\ \{s; \textbf{return } e\} \\ I \vdash \sigma, \overline{a} \Downarrow \overline{v} \qquad I \vdash [\overline{x} \mapsto \overline{v}], s \Downarrow \sigma' \qquad I \vdash \sigma', e \Downarrow r \end{array}}{I \vdash \sigma, y := f(\overline{a}) \Downarrow [y \mapsto r]\sigma}\ \text{ECall}$$

We use $\Downarrow$ for the evaluation relation of both expressions and statements; $\sigma, e \Downarrow \sigma'$ holds when executing e on state σ can result in state σ'. Since programs may be nondeterministic, there may be multiple final states related to a single initial state for a given program. Note that havoc is the only source of nondeterminism when evaluating a FunIMP program. The remaining evaluation rules for FunIMP are standard and can be found in the extended version of the paper [15].

3 Approximating FunIMP Behaviors

In order to modularly reason about relational $\forall\exists$ properties, we first present semantics for capturing the possible executions of a FunIMP program in *any* valid implementation context. In order to account for both "for all" and "there exists" behaviors of functions, we rely on two kinds of specifications. To reason about *all* possible executions of a valid implementation, i.e., a standard *safety* property, we use a *universal* specification. For guarantees about the *existence* of certain executions, we use an *existential* specification.

3.1 Universal Executions

Both kinds of specifications are parameterized over an assertion language $\mathcal{A}$ on program states and a mechanism for judging when a state satisfies an assertion. We write $\sigma \models P$ to denote that a state σ satisfies the assertion P. The universal specifications used to reason about programs on the "for all" side of $\forall\exists$ properties are written as $FA ::= \mathsf{ax}_\forall\ f(\overline{x})\ \{P\}\{Q\}$, where $P \in \mathcal{A}$ is a precondition with free variables in $\overline{x}$ and $Q \in \mathcal{A}$ is a postcondition with free variables in $\overline{x} \cup \{\rho\}$. The postcondition uses the distinguished variable ρ to refer to the value returned by f. Universal specifications promise client programs that the valid implementations of a function will only evaluate to states satisfying the postcondition when evaluated in a starting state that satisfies the precondition.

Definition 1 ($\forall$ – *Compatibility*)**.** *A function definition* $\textsf{def}\ f(\overline{x})\{s; \textsf{return}\ r\}$ *is* $\forall$*-compatible with a universal specification* $\textsf{ax}_\forall\ f(\overline{x})\{P\}\{Q\}$ *if only values satisfying* Q *may be returned whenever* f *is called with arguments satisfying* P*:*

$$\forall \sigma, \sigma'.\ (\sigma \models P)\ \wedge\ (I \vdash \sigma, s \Downarrow \sigma')\ \wedge\ (\sigma', r \Downarrow v) \implies ([\rho \mapsto v]\sigma \models Q)$$

We say that an implementation context I is $\forall$-compatible with a context of universal specifications $S_\forall \in \mathcal{N} \rightarrow FA$ when every definition in I is $\forall$-compatible with the corresponding specification in $S_\forall$.

To characterize the set of possible behaviors of a program under any $\forall$-compatible implementation context, we define a new *overapproximate* semantics for FunIMP, $\Downarrow_\forall$. The evaluation rules of this semantics are based on $\Downarrow$, but they use a universal specification context, $S_\forall$, instead of an implementation context, and replace ECall with the following two evaluation rules:

$$\frac{\begin{array}{c} S_\forall(f) = \textsf{ax}_\forall\ f(\overline{x})\,\{P\}\,\{Q\} \\ S_\forall \vdash \sigma, \overline{a} \Downarrow_\forall \overline{v} \qquad [\overline{x} \mapsto \overline{v}] \models P \qquad [\rho \mapsto r, \overline{x} \mapsto \overline{v}] \models Q \end{array}}{S_\forall \vdash \sigma, y := f(\overline{a}) \Downarrow_\forall [y \mapsto r]\sigma}\ \textsc{ECall}_{\forall 1}$$

$$\frac{S_\forall(f) = \textsf{ax}_\forall\ f(\overline{x})\,\{P\}\,\{Q\} \qquad S_\forall \vdash \sigma, \overline{a} \Downarrow_\forall \overline{v} \qquad [\overline{x} \mapsto \overline{v}] \not\models P}{S_\forall \vdash \sigma, y := f(\overline{a}) \Downarrow_\forall [y \mapsto r]\sigma}\ \textsc{ECall}_{\forall 2}$$

The first rule states that if a function is called with arguments satisfying its precondition, it will return a value satisfying its postcondition; otherwise, the second rule states that it can return *any* value. The latter case allows the overapproximate semantics to capture evaluations where a function is called with arguments that do not meet its precondition. The extended version of this paper [15] includes a complete listing of the $\Downarrow_\forall$ relation.

Any final state of a program evaluated under an implementation context I which is $\forall$-compatible with $S_\forall$ can also be produced using $\Downarrow_\forall$ and $S_\forall$. Appealing to this intuition, we call the evaluations of a FunIMP program p using $\Downarrow_\forall$ the *overapproximate executions* of p under $S_\forall$.

Theorem 1. *When run under an implementation context* I *that is* $\forall$*-compatible with specification context* $S_\forall$ *and an initial state* σ*, a program* p *will either diverge or evaluate to a state* σ' *which is also the result of one of its overapproximate executions under* $S_\forall$*.*

3.2 Existential Executions

Universal specifications approximate function calls on the "for all" side of $\forall\exists$ properties by constraining what a compatible implementation *can* do. Existential specifications approximate the "there exists" executions by describing the required values a valid implementation *must* be able to return. In order to flexibly capture these behaviors, existential pre- and post-conditions are indexed by a set of *choice variables* $\overline{c} \subseteq \mathcal{V}$. Each instantiation of these variables defines

```
def randB(x) {
  skip;
  return 0
}
```

```
def randB(x) {
  r := havoc;
  while (x ≤ r) do r := r − x end;
  return r }
```

```
def randB(x) {
  r := havoc;
  return r
}
```

Fig. 3. Implementations of a function which returns an integer within a bound.

a particular behavior that an implementation has to exhibit. The syntax for writing an existential specification is: $FE ::= \mathsf{ax}_{\exists}\ f(\overline{x})\ [\overline{c}]\ \{P\}\{Q\}$.

We write $A[x/y]$ to denote the predicate A with all free occurrences of x replaced with y. Intuitively, for any instantiation $\overline{v}$ of choice variables $\overline{c}$, an existential specification requires an implementation to produce at least one value satisfying the specialized postcondition $Q[\overline{v}/\overline{c}]$, when called with arguments that satisfy the corresponding precondition $P[\overline{v}/\overline{c}]$. This intuition is embodied in our notion of compatibility for existential specifications:

Definition 2 (∃-Compatibility). *A function definition* $\textit{\textsf{def}}\ f(\overline{x})\{s; \textit{\textsf{return}}\ r\}$ *is ∃-compatible with an existential specification* $\mathsf{ax}_{\exists}\ f(\overline{x})[\overline{c}]\{P\}\{Q\}$ *if, for every selection of choice variables* $\overline{v}$*, calling* f *with arguments that satisfy* $P[\overline{v}/\overline{c}]$ *can return at least one value satisfying* $Q[\overline{v}/\overline{c}]$*:*

$$\forall \sigma, \overline{v}.\ (\sigma \models P[\overline{v}/\overline{c}]) \implies \exists \sigma'.\ (I \vdash \sigma, s \Downarrow \sigma')\ \wedge\ (\sigma', r \Downarrow v) \wedge ([\rho \mapsto v]\sigma \models Q[\overline{v}/\overline{c}])$$

Example 1. To see how universal and existential specifications work together to describe a function's behavior, consider a function $\mathsf{randB}(\mathsf{x})$ which is intended to return some integer between 0 and its argument x. We can write a universal specification requiring all return values to be within the desired bound: $\mathsf{ax}_{\forall}$ $\mathsf{randB}(x)$ $\{0 < x\}$ $\{0 \leq \rho < x\}$. This does not, however, *guarantee* every value in this range is possible. To express this requirement, we reify the choice of the random value using an existential specification: $\mathsf{ax}_{\exists}$ $\mathsf{randB}(x)$ $[c]$ $\{0 < x \wedge 0 \leq c < x\}$ $\{\rho = c\}$. Figure 3 lists a variety of possible randB implementations; the first implementation is compatible with the aforementioned universal specification and the third definition is compatible with the existential specification, but only the middle one satisfies both. Note how c acts as a ghost variable which constrains the choice of the random number. Thus, when reasoning about a client of randB, we can select a concrete value for c that forces the desired result.

Equipped with a context of existential specifications $S_{\exists} \in \mathcal{N} \to FE$, we characterize the set of behaviors a program *must* exhibit under every ∃-compatible implementation context via an underapproximate semantics for FunIMP programs. The judgements of this semantics are denoted as $S_{\exists} \vdash \sigma, p \Downarrow_{\exists} \Sigma$, which reads as: under context $S_{\exists}$ and initial state σ, the program p will produce at least one final state in the set of states Σ. The evaluation rules of this semantics are given in Fig. 4. Most of the rules in Fig. 4 adapt the FunIMP evaluation rules to account for the fact that commands now produce *sets* of states from an

$$\frac{}{S_\exists \vdash \sigma, \textsf{skip} \Downarrow_\exists \{\sigma\}}\ \textsc{ESkip}_\exists \qquad \frac{}{S_\exists \vdash \sigma, x := \textsf{havoc} \Downarrow_\exists \{\sigma' \mid \exists v.[x \mapsto v]\sigma'\}}\ \textsc{EHavoc}_\exists$$

$$\frac{\sigma, a \Downarrow v}{S_\exists \vdash \sigma, x := a \Downarrow_\exists \{[x \mapsto v]\sigma\}}\ \textsc{EAssn}_\exists \qquad \frac{S_\exists \vdash \sigma, s \Downarrow_\exists \Sigma \qquad \Sigma \subseteq \Sigma'}{S_\exists \vdash \sigma, s \Downarrow_\exists \Sigma'}\ \textsc{EConsq}_\exists$$

$$\frac{S_\exists \vdash \sigma, s_1 \Downarrow_\exists \Sigma \qquad \forall \sigma' \in \Sigma.\ S_\exists \vdash \sigma', s_2 \Downarrow_\exists \Sigma'}{S_\exists \vdash \sigma, s_1;\ s_2 \Downarrow_\exists \Sigma'}\ \textsc{ESeq}_\exists$$

$$\frac{\begin{array}{c}\sigma, b \Downarrow \textsf{true} \qquad S_\exists \vdash \sigma, c \Downarrow_\exists \Sigma \\ \forall \sigma' \in \Sigma.\ S_\exists \vdash \sigma', \textsf{while}\ b\ \textsf{do}\ c\ \textsf{end} \Downarrow_\exists \Sigma'\end{array}}{S_\exists \vdash \sigma, \textsf{while}\ b\ \textsf{do}\ c\ \textsf{end} \Downarrow_\exists \Sigma'}\ \textsc{ELpT}_\exists$$

$$\frac{\sigma, b \Downarrow \textsf{false}}{S_\exists \vdash \sigma, \textsf{while}\ b\ \textsf{do}\ c\ \textsf{end} \Downarrow_\exists \{\sigma\}}\ \textsc{ELpF}_\exists \qquad \frac{\sigma, b \Downarrow \textsf{true} \qquad S_\exists \vdash \sigma, s_1 \Downarrow_\exists \Sigma}{S_\exists \vdash \sigma, \textsf{if}\ b\ \textsf{then}\ s_1\ \textsf{else}\ s_2 \Downarrow_\exists \Sigma}\ \textsc{EIfT}_\exists$$

$$\frac{\sigma, b \Downarrow \bot \qquad S_\exists \vdash \sigma, s_2 \Downarrow_\exists \Sigma}{S_\exists \vdash \sigma, \textsf{if}\ b\ \textsf{then}\ s_1\ \textsf{else}\ s_2 \Downarrow_\exists \Sigma}\ \textsc{EIfF}_\exists$$

$$\frac{S_\exists(f) = \textsf{ax}_\exists\ f(\overline{x})\,[c]\,\{P\}\,\{Q\} \qquad S_\exists \vdash \sigma, \overline{a} \Downarrow \overline{v} \qquad [\overline{x} \mapsto \overline{v}] \models P[\overline{k}/\overline{c}]}{S_\exists \vdash \sigma, y := f(\overline{a}) \Downarrow_\exists \qquad \{\sigma' \mid \exists r.\ \sigma' = [y \mapsto r]\sigma\ \wedge\ [\rho \mapsto r, \overline{x} \mapsto \overline{v}] \models Q[\overline{k}/\overline{c}]\}}\ \textsc{ECall}_\exists$$

Fig. 4. The existential evaluation relation.

initial state. For example, the evaluation rule for sequences, ESEQ$_\exists$, states that s_2 produces a final state corresponding to every state in the set produced by s_1. The rule for function calls, ECALL$_\exists$, is the most interesting: it *chooses* one of the behaviors guaranteed by the existential specification of a function and produces a *set* of final states for every return value consistent with that choice.

Every set of final states for a program p produced by these semantics under $S_\exists$ includes a possible final state of p when evaluated under any ∃-compatible implementation context. For this reason, we term the evaluations of p using $\Downarrow_\exists$ the *underapproximate executions* of p under $S_\exists$.

Theorem 2. *If there is an underapproximate evaluation of program p to a set of states Σ from an initial state σ under $S_\exists$, then p must terminate in at least one final state $\sigma' \in \Sigma$ when it is run from σ under an implementation context I that is ∃-compatible with $S_\exists$.*

3.3 Approximating ∀∃ Behaviors

Taken together, the over- and under-approximate semantics allow us to relate the ∀∃ behaviors of multiple client programs under every ∀- and ∃-compatible implementation context. This admits a modular reasoning principle, where if a set of clients can be shown to exhibit some behaviors using the overapproximate and underapproximate semantics, linking the client with any compatible environment will continue to exhibit those behaviors. The key challenge to ensuring these $\forall\exists$ behaviors is identifying, for every overapproximate execution, an appropriate selection of choice variables that cause the underapproximate executions to evaluate to a collection of final states satisfying a desired $\forall\exists$ property.

Example 2. Consider the second example from the introduction, and assume that `randB` has the universal and existential specifications from Example 1. To ensure that `encode` does not reveal anything about its secret input via its public output, it suffices to establish that for any universal execution of `encode` on a specific input, every other possible input to `encode` could produce the same encoded message under the existential semantics. The first execution begins with the statement **`int`** $\texttt{key}^{\texttt{H}}_{\forall}$`= randB(MAX_INT)` (for convenience, we annotate program variables from the first and second executions with the subscripts $\forall$ and $\exists$, respectively). By ECALL$_{\forall 1}$, this statement will update $\texttt{key}^{\texttt{H}}_{\forall}$ to hold a value between 0 and `MAX_INT`. The function then encodes the message using this key, and returns the result. In order to show this leaks nothing, we need to establish a corresponding execution of `encode` that returns this same result regardless of the value of its argument. In effect, this amounts to finding a strategy for instantiating the choice variable in ECALL$_{\exists}$ to assign an appropriate value to $\texttt{key}^{\texttt{H}}_{\exists}$. In this case, the choice is straightforward: we need a c such that c `xor` $\texttt{msg}^{\texttt{H}}_{\exists}$ $=\texttt{enc}^{\texttt{L}}_{\forall}$. Using $\texttt{msg}^{\texttt{H}}_{\exists}$ `xor` $\texttt{enc}^{\texttt{L}}_{\forall}$ for c in ECALL$_{\exists}$ achieves the desired result. Using this strategy, we can construct an appropriate execution in response to *every* execution of `encode`. In contrast, if our existential specification were $\mathsf{ax}_{\exists}$ $\mathsf{randB}(x)$ $[\,]$ $\{0 < x\}$ $\{0 \leq \rho < x\}$, it would only guarantee the existence of a single result, and there would be no workable strategy. Indeed, the first definition of `randB` in Fig. 3 satisfies this specification, and `encode` will always leak the full message when using this implementation!

4 RHLE

We now present RHLE, a relational program logic for proving that a collection of FUNIMP programs exhibit some desired set of $\forall\exists$ behaviors. As a consequence of Theorem 1 and Theorem 2, this entails that properties established in RHLE will continue to hold when the programs are linked with any compatible implementation context.

RHLE specifications use *relational* assertions (denoted $\Phi, \Psi \in \mathcal{A}$) to relate the execution of multiple programs. As normal assertions are predicates on a single state, a relational assertion is a predicate on multiple states. Each program in a RHLE triple operates over a distinct state space. To disambiguate between variables that occur in multiple copies, shared variable names are annotated with an identifier unique to each program. Following existing convention [9,36], we use a natural number to identify which state a variable belongs to. As an example, the relational assertion $x_1 \leq x_2$ is a binary predicate over (at least) two states. This assertion is satisfied by any set of two (or more) states where the value of x in the first state is less than or equal to the value of x in the second.

Table 1. Example RHLE assertions. In the second row, low_x refers to the low security state in program p_x; note the ∀∃ relationship must hold for *any* pair of initial high security values, so $high_x$ is not constrained in the precondition.

Property	RHLE Assertion
Refinement	$S_\forall,\ S_\exists \models \langle \overline{x_1} = \overline{x_2} \rangle\, y_1 := f(\overline{x_1}) \sim_\exists y_2 := f(\overline{x_2})\, \langle y_1 = y_2 \rangle$
Noninterference	$S_\forall,\ S_\exists \models \langle low_1 = low_2 \rangle\, p_1 \sim_\exists p_2\, \langle low_1 = low_2 \rangle$
Injectivity	$S_\forall,\ S_\exists \models \langle x_1 \neq x_2 \rangle\, y_1 := f(x_1) \circledast y_2 := f(x_2) \sim_\exists \mathsf{skip}\, \langle y_1 \neq y_2 \rangle$
Nondeterminism	$S_\forall,\ S_\exists \models \langle x_1 = x_2 \rangle\, \mathsf{skip} \sim_\exists y_1 := f(x_1) \circledast y_2 := f(x_2)\, \langle y_1 \neq y_2 \rangle$

RHLE triples have the form $S_\forall, S_\exists \models \langle \Phi \rangle\, \overline{p_\forall} \sim_\exists \overline{p_\exists}\, \langle \Psi \rangle$ and assert that *for all* universal executions of the programs $\overline{p_\forall}$, *there exist* existential executions of the programs $\overline{p_\exists}$ satisfying the relational pre- and post-condition Φ and Ψ:

$$\begin{aligned} S_\forall, S_\exists \models \langle \Phi \rangle\, \overline{p_\forall} \sim_\exists \overline{p_\exists}\, \langle \Psi \rangle \equiv \forall \overline{\sigma_\forall}\ \overline{\sigma_\exists}\ \overline{\sigma'_\forall}.\ \overline{\sigma_\forall},\ \overline{\sigma_\exists} \models \Phi \ \land\ S_\forall \vdash \overline{\sigma_\forall},\ \overline{p_\forall} \Downarrow_\forall \overline{\sigma'_\forall} \implies \\ \exists \Sigma.\ S_\exists \vdash \overline{\sigma_\exists},\ \overline{p_\exists} \Downarrow_\exists \Sigma \ \land\ \forall \sigma'_\exists \in \Sigma.\ \overline{\sigma'_\forall},\ \overline{\sigma'_\exists} \models \Psi \end{aligned}$$

We use $\circledast$ to delineate different programs on the universal and existential sides of $\sim_\exists$ so that, e.g., a sequence of n programs $\overline{p}$ is also denoted as $p_1 \circledast \ldots \circledast p_n$. For example, to assert the program $x := \mathsf{havoc}$ is nondeterministic, we write a RHLE triple with two copies of the program, adding a subscript to the variable x in each for clarity: $\cdot \models \langle \top \rangle\, \mathsf{skip} \sim_\exists x_1 := \mathsf{havoc} \circledast x_2 := \mathsf{havoc}\, \langle x_1 \neq x_2 \rangle$. This triple says that, for all starting states and all executions of the trivial program skip, there exist executions of the programs $x_1 := \mathsf{havoc}$ and $x_2 := \mathsf{havoc}$ such that $x_1 \neq x_2$ after both programs have executed. Note that $\circledast$ is *not* a concatenation operator; it does nothing more than delineate multiple programs in a RHLE triple. Table 1 gives some additional examples of RHLE assertions.

$$\frac{}{S_\forall, S_\exists \vdash \langle \Phi \rangle\ \overline{\text{skip}} \sim_\exists \overline{\text{skip}}\ \langle \Phi \rangle}\ \textsc{Finish} \qquad \frac{S_\forall, S_\exists \vdash \langle \Phi \rangle\ \overline{p_\forall; \mathsf{skip}} \sim_\exists \overline{p_\exists; \mathsf{skip}}\ \langle \Psi \rangle}{S_\forall, S_\exists \vdash \langle \Phi \rangle\ \overline{p_\forall} \sim_\exists \overline{p_\exists}\ \langle \Psi \rangle}\ \textsc{SkipI}$$

$$\frac{\begin{array}{c} \forall \sigma\ \overline{\sigma_\exists}.\ S_\forall \vdash \{\Phi \mid_i \sigma,\ \overline{\sigma_\exists}\}\ \ s_i\ \ \{\Phi' \mid_i \sigma,\ \overline{\sigma_\exists}\} \\ S_\forall, S_\exists \vdash \langle \Phi' \rangle\ p_1 \circledast \ldots \circledast s'_i \circledast \ldots \circledast s_n \sim_\exists \overline{p_\exists}\ \langle \Psi \rangle \end{array}}{S_\forall, S_\exists \vdash \langle \Phi \rangle\ p_1 \circledast \ldots \circledast s_i;\ s'_i \circledast \ldots \circledast p_n \sim_\exists \overline{p_\exists}\ \langle \Psi \rangle}\ \textsc{Step}\forall$$

$$\frac{\begin{array}{c} \forall \overline{\sigma_\forall}\ \sigma.\ S_\exists \vdash [\Phi \mid_i \overline{\sigma_\forall},\ \sigma]\ \ s_i\ \ [\Phi' \mid_i \overline{\sigma_\forall},\ \sigma]_\exists \\ S_\forall, S_\exists \vdash \langle \Phi' \rangle\ \overline{p_\forall} \sim_\exists p_1 \circledast \ldots \circledast s'_i \circledast \ldots \circledast p_n\ \langle \Psi \rangle \end{array}}{S_\forall, S_\exists \vdash \langle \Phi \rangle\ \overline{p_\forall} \sim_\exists p_1 \circledast \ldots \circledast s_i;\ s'_i \circledast \ldots \circledast p_n\ \langle \Psi \rangle}\ \textsc{Step}\exists$$

Fig. 5. Core RHLE proof rules.

The core logic of RHLE is given in Fig. 5. Relational proofs are built by reasoning about the topmost statement of either one of the universally quantified programs via the Step∀ rule or one of the existentially quantified programs using the Step∃ rule. Once all program statements have been considered, final proof obligations can be discharged using the Finish rule. The SkipI rule is used to ensure that all programs end with skip, so that Finish can be applied. Both Step rules rely on non-relational logics for reasoning about the universal $S_\forall \vdash \{P\}\ p\ \{Q\}$ and existential $S_\exists \vdash [P]\ p\ [Q]_\exists$ behaviors of single statements; we will present the details of both logics shortly. The Step rules employ a projection operation, $\overline{\sigma}|_i\Psi$, which maps a relational assertion to a non-relational one. Given a collection of n states, $\Psi|_i\overline{\sigma}$ is satisfied by any state σ' which satisfies Ψ when inserted at the ith position:

$$\sigma' \models \Psi\,|_i\,\overline{\sigma} \quad \equiv \quad \sigma_1, \ldots, \sigma_{i-1}, \sigma', \sigma_{i+1}, \ldots, \sigma_n \models \Psi$$

In effect, this operation ensures the states of the other programs remain unchanged when reasoning about the ith program in the triple.

Universal Hoare Logic. The program logic for universal executions has a standard partial correctness semantics:

$$S_\forall \models \{P\}\ p\ \{Q\} \quad \equiv \quad \forall \sigma, \sigma'.\ \sigma \models P \wedge S_\forall \vdash \sigma, p \Downarrow_\forall \sigma' \implies \sigma' \models Q$$

The rules of this logic are largely standard[1], except for the rule for function calls, which uses a context of universal function specifications:

$$\frac{S_\forall(f) = \mathsf{ax}_\forall\ f(\overline{x})\{P\}\{Q\}}{S_\forall \vdash \left\{\begin{array}{c} P[\overline{a}/\overline{x}] \quad \wedge \\ \forall v.Q[v/\rho; \overline{a}/\overline{x}] \quad \implies \quad R[v/y] \end{array}\right\}\ y := f(\overline{a})\ \{R\}}\ \forall\text{Spec}$$

Existential Hoare Logic. The assertions of our program logic for existential executions say that, for any state meeting the precondition, there *exists* an execution of the program ending in a set of states meeting the post-condition:

$$S_\exists \models [P]\ p\ [Q]_\exists \quad \equiv \quad \forall \sigma.\ \sigma \models P \implies \exists \Sigma.\ S_\exists \vdash \sigma, p \Downarrow_\exists \Sigma \quad \wedge \quad \forall \sigma' \in \Sigma.\ \sigma' \models Q$$

These rules are largely standard *total* Hoare logic rules[2], augmented with a rule for calls to existentially specified functions:

$$\frac{S_\exists(f) = \mathsf{ax}_\exists f(\overline{x})\ [\overline{c}]\ \{P\}\ \{Q\}}{S_\exists \vdash \left[\begin{array}{l} \exists \overline{k}.\ ([\overline{x} \mapsto \overline{a}] \models P[\overline{k}/\overline{c}] \\ \quad \wedge \quad \exists v.[\rho \mapsto v, \overline{x} \mapsto \overline{a}] \models Q[\overline{k}/\overline{c}] \\ \quad \wedge \quad \forall v.[\rho \mapsto v, \overline{x} \mapsto \overline{a}] \models Q[\overline{k}/\overline{c}] \\ \qquad\qquad \implies R[v/y]) \end{array}\right]\ y := f(\overline{a})\ [R]_\exists}\ \exists\text{Spec}$$

[1] The extended version of this paper [15] gives a full listing of the rules of this logic.
[2] The full existential logic is presented in the extended version of this paper [15].

The precondition of this rule is quantified over instantiations $\overline{k}$ of the specification's choice variables. The first of the three conjuncts under this quantifier ensures that the statement is executed in a state satisfying the function's precondition. The next conjunct ensures that the function's post-condition is inhabited. The final conjunct requires that every possible return value satisfying the function's post-condition also satisfies the triple's post-condition.

Example 3. Given the existential specification $\mathsf{ax}_\exists$ $\mathsf{zeroOrOne}()$ $[c]$ $\{c = 0 \vee c = 1\}$ $\{\rho = c\}$, we can use ∃SPEC (along with the standard rule for while loops, see the extended paper [15]) to prove the existential assertion $S_\exists \vdash [k = 0]$ $\mathsf{while}\ k < 4\ \mathsf{do}\ k := k + \mathsf{zeroOrOne}()\ \mathsf{end}$ $[k = 4]_\exists$. This loop *could* loop forever by choosing to add 0 to k at every iteration. Nevertheless, by using measure $4 - k$ with the well-founded relation $<$ and instantiating the choice variable with 1 at each iteration, we can prove a terminating path through the program exists.

4.1 Synchronous Rules

While the rules in Fig. 5 are sufficient to reason about relational properties, it is possible to lessen the verification burden for structurally similar programs by employing *synchronous rules* which exploit structural similarities between the programs being verified [27]. Reasoning over similar control flow structures in lockstep can reduce the space of states verification must consider and simplify loop invariants. This is particularly useful when reasoning about *hyperproperties* [12], or relational properties on multiple executions of the *same* program. In order to more easily reason about structurally similar programs, RHLE also includes synchronous rules inspired by the Cartesian loop logic presented by Sousa and Dillig [36]. The extended version of this paper [15] includes a full listing of these rules.

Example 4. Consider proving that $\mathsf{while}\ (\mathsf{x} < 10)\ \mathsf{do}\ \mathsf{y} := \mathsf{y} + \mathsf{randB}(9)\ \mathsf{end}$ refines $\mathsf{while}\ (\mathsf{x} < 10)\ \mathsf{do}\ \mathsf{y} := \mathsf{y} + \mathsf{randB}(5);\ \mathsf{y} := \mathsf{y} + \mathsf{randB}(6)\ \mathsf{end}$. Intuitively, the first program refines the second because the bodies of the loops are themselves refinements. A proof using only the rules in Fig. 5 is unable to take advantage of this intuition, however. Instead, the proof requires a sufficiently strong invariant characterizing the behavior of the entire loop on the left, and then an invariant for the righthand program that accounts for the behavior of individual iterations of the lefthand loop.

The SYNCLOOPS rule is designed for this situation:

$$\frac{\begin{array}{c} S_\forall, S_\exists \vdash \langle \mathbb{I} \wedge \bigwedge_{0 \le i \le n} b_i \rangle\ s_0 \circledast \cdots \circledast s_k \sim_\exists s_{k+1} \circledast \cdots \circledast s_n\ \langle \mathbb{I} \rangle \\ \mathbb{I} \wedge \bigwedge_{0 \le i \le n} \neg b_i \implies \Psi \qquad \mathbb{I} \wedge \neg \bigwedge_{0 \le i \le n} b_i \implies \bigwedge_{0 \le i \le n} \neg b_i \end{array}}{\begin{array}{c} S_\forall, S_\exists \vdash \langle \mathbb{I} \rangle\ \mathsf{while}\ b_0\ \mathsf{do}\ s_0\ \mathsf{end} \circledast \cdots \circledast \mathsf{while}\ b_k\ \mathsf{do}\ s_k\ \mathsf{end} \\ \sim_\exists \mathsf{while}\ b_{k+1}\ \mathsf{do}\ s_{k+1}\ \mathsf{end} \circledast \cdots \circledast \mathsf{while}\ b_n\ \mathsf{do}\ s_n\ \mathsf{end}\ \langle \Psi \rangle \end{array}}\ \textsc{SyncLoops}$$

The first premise of this rule says that executing all loop bodies preserves some invariant $\mathbb{I}$, the second ensures the invariant is strong enough to imply the post-condition, and the third requires all loops to end on the same iteration. Since this invariant is reestablished after the execution of every loop body; the invariant that y_1 and y_2 are equal at each iteration suffices to verify this example.

4.2 Soundness

The combination of the core and synchronous rules provide a sound methodology for reasoning about $\forall\exists$ properties:

Theorem 3 (RHLE is Sound). *Suppose $S_\forall, S_\exists \vdash \langle \Phi \rangle\ \overline{p_\forall} \sim_\exists \overline{p_\exists}\ \langle \Psi \rangle$. Then, for any function context I compatible with $S_\forall$ and $S_\exists$, any set of initial states $\overline{\sigma_\forall}$ and $\overline{\sigma_\exists}$ satisfying Φ, and every collection of final states $\overline{\sigma'_\forall}$ of $\overline{p_\forall}$, there must exist a collection of final states produced by $\overline{p_\exists}$ that, together with $\overline{\sigma'_\forall}$, satisfies the relational post-condition Ψ.*

5 Verification

```
Algorithm 1: RHLEVerify
  Inputs : $\Phi$, relational precondition
           $p_\forall$, universal programs
           $p_\exists$, existential programs
           $\Psi$, relational postcondition
  Output: $\langle\Phi\rangle\ p_\forall \sim_\exists p_\exists\ \langle\Psi\rangle$ validity
1 begin
2     $\overline{\Psi} \leftarrow (\varnothing, \varnothing, \Psi)$
3     $(\overline{a}, \overline{e}, \Psi') \leftarrow$
        VCGen($\overline{skip; p_\forall}, \overline{skip; p_\exists}, \overline{\Psi}$)
4     return
        Verify($\forall\overline{a}\exists\overline{e}.\ \Phi \implies \Psi'$)
```

We now turn to the relational verification algorithm based on RHLE, presented in Algorithm 1. The algorithm is implicitly parameterized over a pair of universal and existential contexts, and `Verify`, a decision procedure for checking validity of a formula in the underlying assertion logic. The bulk of the work is delegated to `VCGen`, presented in Algorithm 2, which builds a weakest relational precondition for the input RHLE triple. The algorithm then checks that the RHLE triple's precondition entails the calculated weakest precondition.

The body of `VCGen` builds a formula by recursively generating verification conditions for the input programs statement by statement. This loop tries to maximize opportunities to apply synchronous rules at each step, as these rules allow us to simultaneously generate proof obligations for multiple subprograms, as discussed in Sect. 4.1. After establishing there are still program statements to step over (lines 3–4), `VCGen` looks for and processes any trailing program statements which are not loops (lines 5–8), as such statements are not subject to synchronous rule applications. To process individual program statements, `VCGen` relies on a pair of verification condition generators, $\mathtt{VC}_\forall$ and $\mathtt{VC}_\exists$, for the non-relational program logics. These functions are largely standard weakest

Algorithm 2: VCGen

```
Inputs : $\overline{p_\forall}$, a set of universal programs
         $\overline{p_\exists}$, a set of existential programs
         $\overline{\Psi} = (Q_\forall, Q_\exists, \Psi)$, $\Psi$ a postcondition with quantified variables $Q_\forall$, $Q_\exists$
Output: $(\{v_0, \ldots, v_n\}, \{w_0, \ldots, w_n\}, \Phi)$ such that $v_i$, $w_i$ free in $\Phi$ and
         $\langle\Phi\rangle\ \overline{p_\forall} \sim_\exists \overline{p_\exists}\ \langle\Psi\rangle$ is valid if $\forall v_0, \ldots, v_n\ \exists w_0, \ldots w_n.\ \Phi \implies \Psi$
1  begin
2      match $\overline{p_\forall} \sim_\exists \overline{p_\exists}$:
3          case $\overline{skip} \sim_\exists \overline{skip}$ do
4              return $\overline{\Psi}$
5          case $\overline{p'_\forall} \circledast (s_1; s_2) \circledast \overline{p''_\forall} \sim_\exists \overline{p_\exists}$ where $s_2$ not a loop do
6              VCGen($\overline{p'_\forall} \circledast s_1 \circledast \overline{p''_\forall}, \overline{p_\exists}, VC_\forall(s_2, \overline{\Psi})$)
7          case $\overline{p_\forall} \sim_\exists \overline{p'_\exists} \circledast (s_1; s_2) \circledast \overline{p''_\exists}$ where $s_2$ not a loop do
8              VCGen($\overline{p_\forall}, \overline{p'_\exists} \circledast s_1 \circledast \overline{p''_\exists}, VC_\exists(s_2, \overline{\Psi})$)
9          case $\overline{p'_\forall} \circledast s_1;$ if $b$ then $s_t$ else $s_e \circledast \overline{p''_\forall} \sim_\exists \overline{p'_\exists}$ do
10             $(Q_\forall, Q_\exists, \Psi_T) \leftarrow$ VCGen($\overline{p'_\forall} \circledast s_1; s_t \circledast \overline{p''_\forall}, \overline{p_\exists}, b \implies \Psi$))
11             $(Q'_\forall, Q'_\exists, \Psi_E) \leftarrow$ VCGen($\overline{p'_\forall} \circledast s_1; s_e \circledast \overline{p''_\forall}, \overline{p_\exists}, \neg b \implies \Psi$))
12             return $(Q_\forall \cup Q'_\forall, Q_\exists \cup Q_\exists, \Psi_T \wedge \Psi_E)$
13         case $p_0;$ while $b_0$ do $s_0 \circledast \cdots \circledast p_{i-1};$ while $b_{i-1}$ do $s_{i-1} \sim_\exists$
             $p'_i;$ while $b_i$ do $s_i \circledast \cdots \circledast p'_n;$ while $b_n$ do $s_n$ do
14             $\mathbb{I} \leftarrow$ FindInvariant(while $b_0$ do $s_0 \circledast \cdots \circledast$ while $b_{i-1}$ do $s_{i-1} \sim_\exists$
                 while $b_i$ do $s_i \circledast \cdots \circledast$ while $b_n$ do $s_n$)
15             $(Q'_\forall, Q'_\exists, \Psi_{body}) \leftarrow$ VCGen($s_0 \circledast \cdots \circledast s_{i-1} \sim_\exists s_i \circledast \cdots \circledast s_n, \mathbb{I}$)
16             $inductive \leftarrow \mathbb{I} \wedge \bigwedge_{0 \le i \le n} b_i \implies \Psi_{body}$
17             $lockstep \leftarrow \mathbb{I} \wedge \neg \bigwedge_{0 \le i \le n} b_i \implies \bigwedge_{0 \le i \le n} \neg b_i$
18             $post \leftarrow \mathbb{I} \wedge \bigwedge_{0 \le i \le n} \neg b_i \implies \Psi$
19             $(Q_\forall, Q_\exists, \Psi) \leftarrow \overline{\Psi}$
20             if Verify($Q_\forall \cup Q'_\forall,\ Q_\exists \cup Q'_\exists,\ inductive \wedge lockstep \wedge post$) then
21                 VCGen($\overline{p}, \overline{p'}, (Q_\forall, Q_\exists, \mathbb{I})$)
22             else
23                 next case
24         case $\overline{p'_\forall} \circledast (s_1; s_2) \circledast \overline{p''_\forall} \sim_\exists \overline{p_\exists}$ do
25             VCGen($\overline{p'_\forall} \circledast s_1 \circledast \overline{p''_\forall}, \overline{p_\exists}, vc_\forall(s_2, \overline{\Psi})$)
26         case $\overline{p_\forall} \sim_\exists \overline{p'_\exists} \circledast (s_1; s_2) \circledast \overline{p''_\exists}$ do
27             VCGen($\overline{p_\forall}, \overline{p'_\exists} \circledast s_1 \circledast \overline{p''_\exists}, vc_\exists(s_2, \overline{\Psi})$)
```

precondition generators extended with support for existential function calls. The consequents of ∀SPEC and ∃SPEC immediately yield weakest precondition rules, so that if $S_\forall(f) = \mathsf{ax}_\forall\ f(\overline{x})\{P\}\{Q\}$ and $S_\exists(f) = \mathsf{ax}_\exists f(\overline{x})\ [\overline{c}]\ \{P\}\ \{Q\}$, then:

$$\texttt{VC}_\forall(\Psi, y := f(\overline{a})) = P[\overline{a}/\overline{x}] \wedge \forall v.Q[v/\rho; \overline{a}/\overline{x}] \implies \Psi[v/y]$$

$$\texttt{VC}_\exists(\Psi, y := f(\overline{a})) = \exists \overline{k}.\ ([\overline{x} \mapsto \overline{a}] \models P[\overline{k}/\overline{c}]\ \wedge\ \exists v.[\rho \mapsto v, \overline{x} \mapsto \overline{a}] \models Q[\overline{k}/\overline{c}]$$
$$\wedge\ \forall v.[\rho \mapsto v, \overline{x} \mapsto \overline{a}] \models Q[\overline{k}/\overline{c}] \implies \Psi[v/y])$$

If the first three cases fail, the final statements of all the remaining programs are loops. In this case, VCGen attempts to simultaneously process the loops (lines 9–19) à la the SyncLoops rule in Example 4. To be eligible for fusion, loops must execute in lockstep. This condition is checked (line 16) before returning; if loops may execute different numbers of times, the algorithm proceeds to the next match case. If no synchronized reasoning is possible, VCGen defaults to stepping over an arbitrary loop in one of the programs (lines 20–23).

VCGen is parameterized over a procedure called FindInvariant, which acts as an oracle for relational loop invariants. Our prototype implementation of Algorithm 1 currently requires loops to be annotated with their invariants; these annotations are used to implement FindInvariant. We have experimented with adapting both purely logical [18,19] and data-driven approaches [30,31] for invariant inference, but have yet to discover one that is effective for our larger benchmarks. Unlike traditional loop invariants, which must be re-established on every possible execution of the loop body, invariants in existentially quantified executions need only be re-established on a subset of the possible executions of the body. A robust invariant inference approach thus requires finding not only the invariant itself, but a strategy for instantiating choice variables that consistently re-establish the chosen invariant. Scalable invariant inference for existentially quantified executions is an important and interesting direction for future work.

See the extended version of this paper [15] for an example application of Algorithm 1 to RandB.

6 Implementation and Evaluation

To evaluate our approach, we have implemented ORHLE, a publicly available [16] automatic program verifier based on Algorithm 1. ORHLE is implemented in Haskell, and uses Z3 as a backend solver to fill the role of Verify. As previously mentioned, invariants are provided by the programmer via annotations in the code. Input to ORHLE consists of a collection of FunIMP programs, a declaration of how many copies of each program should be included in the universal and existential contexts, and a collection of function specifications expressed using the SMT-LIB2 format. Functions can have both universal and existential specifications, with the latter containing declarations of choice variables. See the extended version of this paper [15] for example ORHLE input listings. ORHLE outputs a set of verification conditions along with a success or failure message. When a property fails to verify, ORHLE outputs a falsifying model.

Our evaluation addresses the following questions:

(R1) Is RHLE *expressive* enough to represent a variety of interesting properties?
(R2) Is our approach *effective*; that is, can it be used to verify or invalidate relational assertions about a diverse corpus of programs?
(R3) Is it possible to realize an *efficient* implementation of our verification approach which returns results within a reasonable time frame?

To answer these questions, we have developed a suite of 41 programs over 5 kinds of relational specifications drawn from the literature. We have also compiled an additional set of 12 benchmarks over two non-relational existentially quantified properties in order to evaluate similar questions about the non-relational existential logic from Sect. 4. Both sets of benchmarks contain a mix of valid and invalid properties. We have made these benchmarks publicly available[3] via GitHub [17].

Our benchmarks for the non-relational existential logic from Sect. 4 fall into two categories:

Winning Strategy. Programs in this category play a simplified version of the card game twenty-one. Players start with two cards valued between 1 and 10, and can then request any number of additional cards. The goal is to get a hand value as close to 21 as possible without going over. The property of interest is whether an algorithmic strategy for this game permits the *possibility* of achieving the maximum hand value of 21 given any starting hand.

Branching Time Properties. Our next set of benchmarks are taken from work by Cook and Koskinen [14] which considered verification of properties of single programs expressed in CTL. The programs in this category are adaptations of the subset of those benchmarks which assert the existence of desirable final states and are thus expressible in RHLE.

Our set of relational benchmarks cover program refinement in addition to:

Noninterference. Generalized noninterference is a possibilistic information security property which ensures that programs do not leak knowledge about high-security state via low-security outputs. Our formalization of this property is based on Mclean [26] and requires that, for any execution of a program p whose state is divided into high security p_H and low security p_L partitions, any other starting state with the same initial low partition can potentially yield the same final low partition, regardless of the high partition.

Delimited Release. Delimited release is a relaxation of generalized noninterference which allows for limited information about secure state to be released. For example, given a confidential list of employee salaries, it may be acceptable to publicize the average salary as long as no other salary information is leaked. We formulate delimited release as a noninterference property with an additional condition requiring that the programs agree on the values of the released information. For the previous example, we would add a precondition asserting the average salary across all executions is equal.

Parameter Usage. Our parameter usage benchmarks check whether a function parameter is semantically unused, in that the existence of the parameter does

[3] Branching time property benchmarks are adapted from a proprietary source, and are thus omitted from the publicly available benchmarks.

Property	**Shape**	**Pos**	**Neg**	**Unk**	**Med**(ms)	**Max**(ms)
Delimited Release	$\forall p_1 \exists p_2$	7	6	0	222	253
Flaky Tests	$\exists p_1 p_2$	2	0	0	231	245
Generalized Noninterference	$\forall p_1 \exists p_2$	4	6	0	222	229
Parameter Usage	$\forall p_1 \exists p_2$	4	3	0	220	245
Program Refinement	$\forall p_1 \exists p_2$	4	4	1	224	1367
Winning Strategy	$\exists p$	1	2	0	228	230
Branching Time	$\exists p$	7	2	0	226	259

Fig. 6. ORHLE verification results over a set of relational and non-relational properties. The **Shape** column gives the execution quantification pattern for the property; each property is of the form $\forall p_0 \ldots p_n \exists q_o \ldots q_n$, where p_i's and q_i's are (possibly empty) sets of executions. The **Pos** and **Neg** columns give the number of benchmarks over which the property holds or does not hold, respectively. The **Unk** column gives the number of benchmarks whose verification conditions could not be decided by the SMT solver. The **Med** and **Max** columns give (respectively) the median and maximum verification times in milliseconds over each set of benchmarks.

not affect the program's reachable final states. For example, the `flag` parameter in `f(flag) = if flag then return 1 else return 1` is syntactically used in `f`, even affecting its control flow, but does not have any effect on `f`'s possible outputs; we therefore consider `flag` to be semantically unused. For an n-ary function $f(p_1, \ldots, p_n)$, we say parameter p_i is semantically unused if

$$\langle v_i \neq w_i \wedge \bigwedge_{j \neq i} v_j = w_j \rangle\ a := f(v_1, \ldots, v_n) \sim_\exists b := f(w_1, \ldots, w_n)\ \langle a = b \rangle$$

Flaky Tests. Tests of program behavior which can nondeterministically pass or fail pose a significant hazard as they can trigger false alarms or allow regressions to go undetected. We modeled representative nondeterministic tests in FunIMP based on examples from The Illinois Dataset of Flaky Tests (IDoFT) [24,35], framing flakiness as a $\forall\exists$ property containing only existential executions. We consider a test verifiably flaky when there exists both a test execution that succeeds and one that fails. We model nondeterminsitic system behavior (e.g., `getCurrentTimeMs()` or the results of network calls) as function calls. For example, to model the imprecision of thread `sleep`s, we give the verifier leeway to sleep within a ±20 ms window around the requested interval: $\mathsf{ax}_\exists$ $\mathsf{sleep}(\text{interval}, \text{currentTime})$ $[\text{sleepTime}]$ $\{0 \leq \text{sleepTime} \ \wedge\ \text{interval} - 20 \leq \text{sleepTime} \leq \text{interval} + 20\}$ $\{\rho = \text{currentTime} + \text{sleepTime}\}$.

The variety of properties we were able to represent in ORHLE provides evidence that it is sufficiently expressive (R1). To show that ORHLE is both effective and efficient (R2)-(R3), we have used it to verify and/or invalidate examples of the benchmark properties described above. All of these experiments were done using an Intel Core i7-6700K CPU with 8 4GHz cores. Figure 6 presents the results of these experiments. ORHLE yielded the expected verification result in all cases except for one refinement benchmark, where the backing SMT solver (Z3) was unable to determine the validity of the verification conditions. While most benchmarks' verification conditions fell within the theory of linear integer arithmetic, verification conditions fell in a non-decidable fragment of arithmetic in this benchmark. This undecidable instance accounts for the outlier maximum verification time in the refinement benchmarks. Overall, these results offer evidence that ORHLE is both effective and efficient for verifying a variety of existential and ∀∃ properties.

7 Related Work

Relational Program Logics. Relational program logics are a common approach to verifying relational specifications. Relational Hoare Logic [9] (RHL) was one of the first examples of these logics, and is capable of proving 2-safety properties. Relational Higher-order Logic [2] is a higher-order relational logic for reasoning about higher-order functional programs expressed in a simply-typed λ-calculus. Probabilistic RHL [8] is a logic for reasoning about probabilistic programs in order to prove security properties of cryptographic schemes. The relational logic closest to RHLE is Cartesian Hoare Logic [36] (CHL) developed by Sousa and Dillig. This logic which provides an axiomatic system for reasoning about k-safety hyperproperties along with an automatic verification algorithm. RHLE can be thought of as an extension of CHL for reasoning about the more general class of ∀∃ properties. Nagasamudram and Naumann [27] examine *alignment completeness* for relational Hoare logics, which classifies the ability of these logics to reason about programs in lockstep. Banerjee et al. [4] introduce a relational Hoare logic capable of reasoning about encapsulation and invariant hiding, but which is confined to 2-safety properties.

Underapproximate Program Logics. Several program logics have been proposed to reason about the existence of particular executions of a single program, similar to the non-relational existential logic presented in Sect. 4. Reverse Hoare Logic [39] is a program logic for reasoning about reachability over single executions of programs which have access to a nondeterministic binary choice ($\sqcup$) operator. Incorrectness Logic [29] is a recent adaptation of Reverse Hoare Logic to a more realistic programming language. While these logics express the existence of a satisfying start state for all satisfying end states ($\forall\sigma'\exists\sigma$), the existential logic presented in Sect. 4 requires there to exist a satisfying end state for all satisfying start states ($\forall\sigma\exists\sigma'$). Reverse Hoare Logic and Incorrectness Logic both reason about reachability over single executions, but properties in these logics are pure underapproximations: every state in a given postcondition must be reachable. In contrast, our reasoning over existential specifications is underapproximate *with respect to the choice variables only.* While every valid choice value must correspond to a reachable set of final states, each of these sets are still overapproximate. This feature of our existential specifications enables a natural integration with standard Hoare logics.

First-order dynamic logic [33] is a reinterpretation of Hoare logic in first-order, multi-modal logic. For a program p, the modal operators $[p]$ and $\langle p\rangle$ capture universal and existential quantification over program executions. Our universal Hoare triple $\vdash \{P\}p\{Q\}$ corresponds to $P \implies [p]Q$, and our existential Hoare triple $\vdash [P]p[Q]_\exists$ corresponds to $P \implies \langle p\rangle Q$. In contrast to RHLE, dynamic logic reasons about properties of single program executions.

Prophecy Variables. Prophecy variables were originally introduced by Abadi and Lamport [1] in order to establish refinement mappings between state machines. Choice variables in our existential specifications are similar to prophecy variables in that they capture the required value of some "future" state, although we use them as part of a program logic rather than to reason about refinement mappings between state machines. Jung et al. [22] incorporate prophecy variables into a separation Hoare logic to reason about nondeterminism in concurrent programs, but differ from our approach in that the program logic operates in a non-relational setting and is designed for interactive and not automated verification.

Relational Verification. The concept of a hyperproperty was originally introduced by Clarkson and Schneider [12], building on earlier work by Terauchi and Aiken [37]. The initial work discusses verification but it does not offer an algorithm; numerous program techniques have been subsequently proposed to verify hyperproperties. Product programs are an alternative approach to relational verification [5]. This approach can leverage existing non-relational verification tools and techniques when verifying the product program, but the large state space of product programs can make verification difficult in practice. Product programs have been used to verify k-safety properties and reason about noninterference and secure information flow [7,23]. Barthe et al. [6] have developed a set of necessary conditions for "left-product programs"; these product programs can be used to verify hyperproperties outside of k-safety, including our $\forall\exists$ properties, although the work does not address how to construct left-product programs.

Unno et al. [38] have developed a technique for verifying $\forall\exists$ properties including program refinement, generalized noninterference, and cotermination by encoding a constraint satisfaction problem expressed using a generalization of constrained Horn clauses. The approach solves constraints using a stratified CEGIS approach, and can synthesize non-trivial alignment predicates for interleaving executions of loop bodies. This work is not based on a Hoare-style program logic, but rather develops per-property embeddings of $\forall\exists$ verification problems in a novel adaptation of constrained Horn clauses.

There are several modal logics which support a style of existential reasoning similar to our existential logic. Temporal logics like HyperLTL and HyperCTL [11] can be used to reason about hyperproperties, although verification tooling [10] is focused on model checking state transition systems rather than program logics. Coenen et al. [13] examine verification and synthesis of computational models using HyperLTL formulas with alternating quantifiers. Cook et al. [14] examine existential reasoning in branching-time temporal logics by way of removing state space until universal reasoning methods can be used. Lamport and Schneider [25] examine using TLA to verify $\forall\exists$ properties including refinement and GNI. While the above approaches are capable of reasoning about the kinds of liveness properties we consider in this paper, they all focus on model checking state transition systems rather than using a Hoare-style logic to reason directly over programs as in our approach.

8 Conclusion

This paper presented RHLE, a novel relational Hoare-style program logic for reasoning about ∀∃ properties. These properties can capture a variety of interesting behaviors of multiple program executions, including program refinement and information flow properties. Key to our logic is a novel form of function specifications which constrain the set of behaviors that a valid implementation of a function *must* exhibit. We have developed an automated verification algorithm based on RHLE, and we demonstrated that an implementation of this algorithm is able to check the validity of a variety of ∀∃ properties over a benchmark suite of programs.

Acknowledgments. We would like to thank Roopsha Samanta for her valuable input on initial drafts of this work. We would also like to thank the anonymous reviewers of this and previous iterations of this paper for their much-appreciated feedback. This research was partially supported by the National Science Foundation under Grant CCF-1755880 and by a grant from the Purdue Research Foundation.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. In: [1988] Proceedings. Third Annual Symposium on Logic in Computer Science, pp. 165–175 (1988)
2. Aguirre, A., Barthe, G., Gaboardi, M., Garg, D., Strub, P.Y.: A relational logic for higher-order programs. Proc. ACM Program. Lang. **1**(ICFP), 21:1–21:29 (Aug 2017)
3. Appel, A.W.: Verified software toolchain. In: Barthe, G. (ed.) Programming Languages and Systems, vol. 6602, pp. 1–17. Springer, Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19718-5_1
4. Banerjee, A., Nagasamudram, R., Naumann, D.A., Nikouei, M.: A relational program logic with data abstraction and dynamic framing. arXiv preprint arXiv:1910.14560 (2019)
5. Barthe, G., Crespo, J.M., Kunz, C.: Relational Verification Using Product Programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_17
6. 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, pp. 29–43. Springer (2013). https://doi.org/10.1007/978-3-642-35722-0_3
7. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Math. Struct. Comput. Sci. **21**(6), 1207–1252 (2011)
8. Barthe, G., Grégoire, B., Zanella Béguelin, S.: Formal certification of code-based cryptographic proofs. SIGPLAN Not. **44**(1), 90–101 (2009)

9. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 14–25. POPL '04, ACM, New York, NY, USA (2004)
10. Clarke, E., Grumberg, O., Long, D.: Verification tools for finite-state concurrent systems. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1993. LNCS, vol. 803, pp. 124–175. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58043-3_19
11. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: International Conference on Principles of Security and Trust, pp. 265–284. Springer (2014). https://doi.org/10.1007/978-3-642-54792-8_15
12. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)
13. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness, pp. 121–139 (07 2019)
14. Cook, B., Koskinen, E.: Reasoning about nondeterminism in programs. In: Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation, pp. 219–230 (2013)
15. Dickerson, R., Ye, Q., Zhang, M.K., Delaware, B.: Rhle: modular deductive verification of relational ∀∃ properties (extended paper) (2020). 10.48550/ARXIV.2002.02904
16. Dickerson, R., Ye, Q., Zhang, M.K., Delaware, B.: ORHLE (2022). https://doi.org/10.5281/zenodo.7058107
17. Dickerson, R., Ye, Q., Zhang, M.K., Delaware, B.: RHLE Benchmarks (2022). https://github.com/rcdickerson/rhle-benchmarks
18. Dillig, I., Dillig, T., Li, B., McMillan, K.: Inductive invariant generation via abductive inference. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications, pp. 443–456. OOPSLA '13, Association for Computing Machinery, New York, NY, USA (2013)
19. Flanagan, C., Leino, K.R.M.: Houdini, an Annotation Assistant for ESC/Java. In: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity, pp. 500–517. FME '01, Springer-Verlag, Berlin, Heidelberg (2001)
20. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
21. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: Rustbelt: securing the foundations of the rust programming language. Proc. ACM Program. Lang. **2**(POPL), 1–34 (dec 2017)
22. Jung, R., et al.: The future is ours: prophecy variables in separation logic. Proc. ACM Program. Lang. **4**(POPL), 1–32 (Dec 2019)
23. Kovács, M., Seidl, H., Finkbeiner, B.: Relational abstract interpretation for the verification of 2-hypersafety properties, pp. 211–222 (11 2013)
24. Lam, W., Oei, R., Shi, A., Marinov, D., Xie, T.: idflakies: a framework for detecting and partially classifying flaky tests. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp. 312–322 (2019)
25. Lamport, L., Schneider, F.B.: Verifying Hyperproperties with TLA. In: 2021 IEEE 34th Computer Security Foundations Symposium (CSF), pp. 1–16. iSSN: 2374–8303 (Jun 2021)

26. McLean, J.: A general theory of composition for a class of "possibilistic" properties. IEEE Trans. Softw. Eng. **22**(1), 53–67 (Jan 1996)
27. Nagasamudram, R., Naumann, D.A.: Alignment completeness for relational hoare logics. In: 2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pp. 1–13 (2021)
28. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. **37**5(1), 271–307 (2007), festschrift for John C. Reynolds's 70th birthday
29. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL) 1–32 (Dec 2019)
30. Padhi, S., Sharma, R., Millstein, T.: Data-driven precondition inference with learned features. ACM SIGPLAN Notices **51**(6), 42–56 (2016)
31. Padhi, S., Sharma, R., Millstein, T.: Loopinvgen: a loop invariant generator based on precondition inference (2017)
32. Poetzsch-Heffter, A., Müller, P.: A Programming Logic for Sequential Java. In: Swierstra, S.D. (ed.) Programming Languages and Systems, vol. 1576, pp. 162–176. Springer, Berlin Heidelberg, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-49099-X_11
33. Pratt, V.R.: Semantical consideration on Floyd-Hoare logic. In: 17th Annual Symposium on Foundations of Computer Science (sfcs 1976), pp. 109–121. IEEE (1976)
34. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, pp. 55–74 (2002)
35. Shi, A., Gyori, A., Legunsen, O., Marinov, D.: Detecting assumptions on deterministic implementations of non-deterministic specifications. In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 80–90 (2016)
36. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 57–69. PLDI '16, ACM, New York, NY, USA (2016)
37. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Hankin, C., Siveroni, I. (eds.) Static Analysis, vol. 3672, pp. 352–367. Springer, Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/11547662_24
38. Unno, H., Terauchi, T., Koskinen, E.: Constraint-Based Relational Verification. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 742–766. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_35
39. de Vries, E., Koutavas, V.: Reverse hoare logic. In: Proceedings of the 9th International Conference on Software Engineering and Formal Methods, pp. 155–171. SEFM'11, Springer-Verlag, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24690-6_12