Relational Verification via Invariant-Guided Synchronization

Qi Zhou
Georgia Institute of Technology
qzhou80@gatech.edu

David Heath
Georgia Institute of Technology
heath.davidanthony@gatech.edu

William Harris
Galois Inc.
wrharris@galois.com

Relational properties describe relationships that hold over multiple executions of one or more programs, such as functional equivalence. Conventional approaches for automatically verifying such properties typically rely on syntax-based, heuristic strategies for finding *synchronization points* among the input programs. These synchronization points are then annotated with appropriate relational invariants to complete the proof. However, when suboptimal synchronization points are chosen the required invariants can be complicated or even inexpressible in the target theory.

In this work, we propose a novel approach to verifying relational properties. This approach searches for synchronization points and synthesizes relational invariants *simultaneously*. Specifically, the approach uses synthesized invariants as a guide for finding proper synchronization points that lead to a complete proof. We implemented our approach as a tool named PEQUOD, which targets Java Virtual Machine (JVM) bytecode. We evaluated PEQUOD by using it to solve verification challenges drawn from the from the research literature and by verifying properties of student-submitted solutions to online challenge problems. The results show that PEQUOD solve verification problems that cannot be addressed by current techniques.

1 Introduction

Relational properties characterize multiple executions of one or more programs [21]. One example of such a property is that a particular program f over integers is monotonic; i.e.,

$$\forall x, y.\ x > y \Rightarrow f(x) > f(y)$$

This property is relational because it is defined over two arbitrary inputs of f (named x and y, respectively). Relational properties can express important problems, such as the equivalence of two programs or the information-flow security of a single program. Therefore, a tool that could automatically verify relational properties would be highly valuable both to program developers and users.

Substantial effort has been directed toward constructing relational *verifiers*, which attempt to prove that given programs satisfy a given relational property. One effective approach attempts to synthesize proofs in Cartesian Hoare Logic [19, 21], which extends Hoare Logic from individual programs to tuples of programs. Such approaches consider the execution of input programs simultaneously, which enables the construction of relational invariants that describe relationships across the programs. By examining the programs together, a verifier can potentially find simpler invariants than if it had attempted to summarize each program and then compared the summaries.

However, synthesizing proofs in such a system adds a critical new dimension to a verifier's design. In particular, a verifier must choose pairs of control locations to relate, in addition to synthesizing sufficiently strong invariants that relate the programs' data when they reach related locations. Such pairs of locations are referred to as *synchronization points* [19]. Intuitively, certain synchronization points can

E. De Angelis, G. Fedyukovich, N. Tzevelekos and M. Ulbrich (Eds.):
Sixth Workshop on Horn Clauses for Verification and Synthesis and
Third Workshop on Program Equivalence and Relational Reasoning.
EPTCS 296, 2019, pp. 28–41, doi:10.4204/EPTCS.296.6

be annotated with simple relational invariants to form proofs because the variables at the related points maintain similar data. Conversely, non-ideal synchronization points may relate locations for which sufficient invariants can be expressed using only complex formulas, or even formulas expressible only in complex theories and logics.

Selecting synchronization points is particularly difficult when a verifier must prove a relational property over programs with loops or recursive procedures. Specifically, finding an ideal set of synchronization points may require the verifier to consider different numbers of iterations for different loops. As an example, a suitable synchronization strategy might be to model two iterations of a loop in one program for every one iteration of a loop in a second program. This highlights that no straightforward solution, such as modeling each loop exactly once, is effective in general.

For the reasons given above, it is clear that finding effective strategies for selecting synchronization points is an important and difficult problem. The effectiveness of a selecting synchronization points depends on the data relationships in the programs and the property. However, existing approaches [19,21] have relied on syntax-driven, heuristic strategies that first find synchronization points and then attempt to annotate the points with relational invariants to complete the proof. The effectiveness of these strategies usually heavily depends on the programs to which they are applied.

In this paper, we propose a general, automatic technique for synthesizing proofs of relational properties. The key feature of our approach is that it searches the spaces of potential synchronization points and their relational invariants *simultaneously*. Our approach iteratively operates on a sequence of bounded under-approximations of input programs; in each bounded under-approximation, each recursive procedure call is only allowed to execute a bounded number of times. In each iteration, our approach attempts to generate a set of proofs that the bounded programs satisfy the given relational property under *all* possible relevant choices of synchronization points. Our approach synthesizes this set of proofs by solving a single system of *Constrained Horn Clauses*. Then, our approach attempts to find *some* proof of the correctness of the bounded under-approximations that can be generalized to form a proof for the original, unbounded programs. If a valid proof is found, then the verifier has validated the given relational property. Otherwise, our approach continues by considering larger under-approximations of the input programs.

We have implemented our approach as an executable tool, named PEQUOD. PEQUOD targets Java Virtual Machine (JVM) bytecode and has been evaluated on 33 benchmarks, consisting of verification challenge problems and student solutions submitted to online coding platforms. Our evaluation indicates that, in a significant set of practical cases, PEQUOD can efficiently verify relational properties beyond the scope of existing techniques.

The rest of this paper is organized as follows. §2 provides an informal overview of our proof system and of PEQUOD, by example. §3 reviews the technical foundations for our work, and §4 presents the proof system and PEQUOD in detail. §5 presents an empirical evaluation of PEQUOD. §6 concludes by comparing our contribution to related work.

2 Overview

In this section, we illustrate our approach by example. We first introduce a pair of programs that compute the same function. We formalize a relational property that these two programs are equivalent as an extended Hoare Logic Triple. Next, we describe how PEQUOD finds a proof of this triple in §2.1.

Fig. 1 contains two programs, named `tri0` and `tri1`, that each compute the nth *triangle number*: i.e., the sum of all natural numbers up to and including n. `tri0` computes this value by direct recursion while

```
1 public static int tri0 (int n) {
2     if (n <= 0) return 0;
3     else return n + tri0(n - 1); }
```

```
1 public static int tri1Aux(int x, int acc) {
2   if (x <= 0) return acc;
3   else return tri1Aux(x - 1, acc + x); }
4 public static int tri1(int n) {
5     return tri1Aux(n, 0); }
```

Figure 1: `tri0` and `tri1`: equivalent programs that, given integer n, compute the nth triangle number.

```
1 public static int tri0_0 (int n) {
2     if (n <= 0) return 0;
3     else return n + tri0_1(n - 1); }
4 public static int tri0_1 (int n) {
5     if (n <= 0) return 0;
6     else return n + tri0_2(n - 1); }
7 public static int tri0_2 (int n) {
8     if (n <= 0) return 0;}
```

```
1 public static int tri1_0(int n) {
2    return tri1Aux_0(n, 0); }
3 public static int tri1Aux_0(int x, int acc) {
4   if (x <= 0) return acc;
5   else return tri1Aux_1(x - 1, acc + x); }
6 public static int tri1Aux_1(int x, int acc) {
7   if (x <= 0) return acc;
8   else return tri1Aux_2(x - 1, acc + x); }
9 public static int tri1Aux_2(int x, int acc) {
10  if (x <= 0) return acc;
```

Figure 2: $\texttt{tri0}_0$ and $\texttt{tri1}_0$ are under-approximations of the input programs.

`tri1` makes use of an auxilliary procedure, `tri1Aux`, which maintains an accumulator. Despite these differences, these two programs compute the same function.

To verify this equivalence, we can construct a relational property that shows that given equal parameters n, `tri0` and `tri1` compute the same output. This property can be represented as a Hoare Logic Triple over a *product* command:

$$\{\texttt{n}_0 = \texttt{n}_1\}\ \texttt{tri0} \times \texttt{tri1}\ \{\mathsf{ret}_0 = \mathsf{ret}_1\}$$

The product command $\texttt{tri0} \times \texttt{tri1}$ can be understood as the command that executes `tri0` and `tri1` simultaneously. A detailed explanation of product commands is given in §4.1. We annotate variables with subscripts 0 or 1 to indicate which program they model. Variables ret_0 and ret_1 are used to model the output of the respective programs. We refer to the proposition $\texttt{n}_0 = \texttt{n}_1$ as the pre-condition, and the proposition $\mathsf{ret}_0 = \mathsf{ret}_1$ as the post-condition. The triple above states that if $\texttt{n}_0 = \texttt{n}_1$ and both `tri0` and `tri1` are executed, then both programs will return the same value. A proof of this triple would prove the equivalence of the two programs.

2.1 Proving Equivalence Automatically

PEQUOD proves this example Hoare Triple in three steps. First, PEQUOD constructs bounded versions of `tri0` and `tri1` that respect an upper bound on the allowed number of recursive procedure calls. In this example, we set this upper bound to three. Fig. 2 lists the bounded programs $\texttt{tri0}_0$ and $\texttt{tri1}_0$. These two program are bounded because each has finitely many execution paths. These two programs under-approximate `tri0` and `tri1` respectively, because their execution paths are a subset of the execution paths in the original programs. $\texttt{tri0}_2$ and $\texttt{tri1Aux}_2$ are incomplete because they do not have else branches in their conditional statement. These branches are assumed to be unreachable in the current under-approximation.

Second, PEQUOD tries to synthesize a set of proofs for a corresponding Hoare Triple over these bounded programs: $\{\texttt{n}_0 = \texttt{n}_1\}\ \texttt{tri0}_0 \times \texttt{tri1}_0\ \{\mathsf{ret}_0 = \mathsf{ret}_1\}$. The key idea is that PEQUOD will find proofs for *all* possible orders of modeling the execution of $\texttt{tri0}_0$ and $\texttt{tri1}_0$. The resulting proofs represent all possible choices of synchronization points of the bounded programs. For example, in a subset of the bounded proofs, PEQUOD arrives at the following intermediate goal:

$$\{\texttt{n}_0 = \texttt{x}_1\}\ \texttt{tri0}_0 \times \texttt{tri1Aux}_0\ \{\mathsf{ret}_0 + \texttt{acc}_1 = \mathsf{ret}_1\}$$

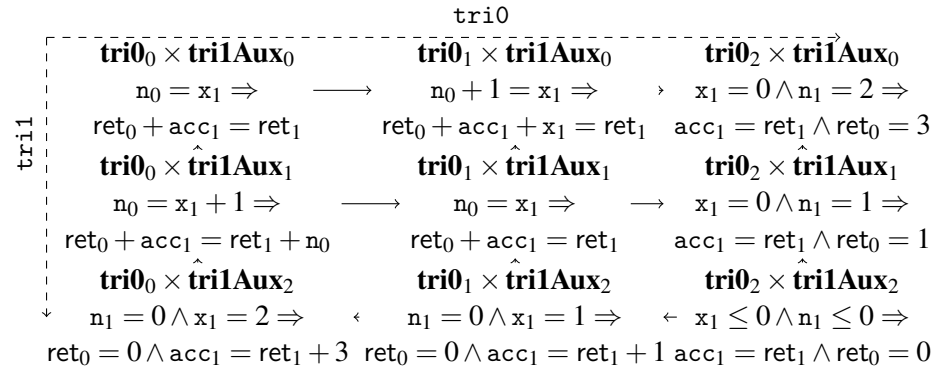

Figure 3: Intermediate products that appear when proving the bounded example program. Each product is depicted with a pre and post invariant that leads to a proof.

PEQUOD continues the proof by either stepping in $\texttt{tri0}_0$ or by stepping in $\texttt{tri1}_0$. 'Stepping' through the product program corresponds to applying particular proof rules that result in new Hoare Triple goals. In other words, PEQUOD proves the triple over $\texttt{tri0}_0 \times \texttt{tri1Aux}_0$ by proving a series of Hoare Triples, which we refer to as a proof path, with proper invariants.

The fact that $\texttt{tri0}_0$ and $\texttt{tri1}_0$ are bounded commands implies that there are finitely many possible proof paths that can be used to prove this bounded goal. Fig. 3 depicts all possible proof paths for a partial proof of this Hoare Triple. The depicted proof is partial because we omit proof goals corresponding to the programs' false branches, for clarity. The upper-leftmost node is the Hoare Triple that PEQUOD must prove. Every proof path over the true branches eventually reaches the the product command $\texttt{tri0}_2 \times \texttt{tri1Aux}_2$ (since $\texttt{tri0}_2$ and $\texttt{tri1Aux}_2$ are under-approximations that allow no further recursion). PEQUOD encodes all possible proof paths into a single set of Constrained Horn Clauses (CHCs), and uses known techniques for solving this system to synthesize proper invariants. In short, PEQUOD uses CHCs to synthesize a set of proofs for all possible proof paths of the bounded program. The method for converting an input Hoare Triple into a CHC system is described in §4.2.2.

Third, PEQUOD attempts to prove the original, unbounded problem by searching for a bounded proof that can be generalized. A suitable approach for solving this example is to consider the synchronization point $\texttt{tri0} \times \texttt{tri1Aux}$, since the data at these two points is highly related. PEQUOD finds this synchronization point automatically and annotates it with appropriate invariants by searching the set of proofs for the bounded programs, as depicted in Fig. 3. PEQUOD begins its search from the top-left node: $\texttt{tri0}_0 \times \texttt{tri1Aux}_0$. In order to find a generalizable proof path, PEQUOD must choose a proof path that passes through the node $\texttt{tri0}_1 \times \texttt{tri1Aux}_1$. Note that the pre-condition and post-condition of two Hoare Triples over these two product commands are the same. Furthermore, the two product commands represent the same unbounded command, $\texttt{tri0} \times \texttt{tri1Aux}$, from the original program. Thus, PEQUOD can use the Hoare Triple over the first command as a hypothesis to prove the second. The details of the proof rule that allows this reasoning is given in §4.1. By choosing this proof path, PEQUOD has also decided to synchronize the two procedures by executing each recursive procedure once. Therefore, PEQUOD finds the proof and synchronization points simultaneously.

Not all proof paths can be generalized to form proofs for the original programs. In fact, any proof path that does not pass through the node $\texttt{tri0}_1 \times \texttt{tri1Aux}_1$ will fail to generalize because **(1)** the relational invariants along the path are inappropriate for use as a hypothesis further down the path and **(2)** the node $\texttt{tri0}_2 \times \texttt{tri1Aux}_2$ cannot be used in the proof because it incompletely models the original programs.

$$\text{E-SKIP}\frac{}{\langle \mathsf{skip}, \sigma\rangle \Downarrow \sigma} \qquad \text{E-ASSIGN}\frac{}{\langle x := e, \sigma\rangle \Downarrow \sigma[x \mapsto e]}$$

$$\text{E-IFTRUE}\frac{\mathsf{eval}(e) = \mathsf{true} \quad \langle c_0, \sigma\rangle \Downarrow \sigma'}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1\ \mathsf{fi}, \sigma\rangle \Downarrow \sigma'} \qquad \text{E-IFFALSE}\frac{\mathsf{eval}(e) = \mathsf{false} \quad \langle c_1, \sigma\rangle \Downarrow \sigma'}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1\ \mathsf{fi}, \sigma\rangle \Downarrow \sigma'}$$

$$\text{E-CALL}\frac{\langle B, \sigma'[\vec{i} \mapsto \vec{x}]\rangle \Downarrow \sigma''}{\langle \vec{x} := N(\vec{r}), \sigma\rangle \Downarrow \sigma[\vec{r} \mapsto \sigma''(\vec{o})]} \qquad \text{E-SEQ}\frac{\langle c_0, \sigma\rangle \Downarrow \sigma' \quad \langle c_1, \sigma'\rangle \Downarrow \sigma''}{\langle c_0\ ;\ c_1, \sigma\rangle \Downarrow \sigma''}$$

Figure 4: Operational semantics of programs in Com.

When PEQUOD cannot find a valid proof for the original programs, PEQUOD increases the bounding number and starts over. The algorithm that generalizes bounded proofs is described in §4.2.3.

3 Background

In this section, we present the technical background for our approach. In §3.1, we formalize the imperative target language. In §3.2, we introduce Constrained Horn Clause (CHC) systems as a class of logic-programming problems.

3.1 Target Language

In this section, we give the formal definition of the target language: an imperative language with conditionals and (possibly recursive) procedures. In order to define a program, we first give the definition of a command. In the following, we use the metavariable x to represent program variables, e to represent program expressions, and N to represent procedure names. The space of commands Com is defined inductively, as follows:

$$\mathsf{Com} ::= \mathsf{skip} \mid x := e \mid \mathsf{if}\ e\ \mathsf{then}\ \mathsf{Com}\ \mathsf{else}\ \mathsf{Com}\ \mathsf{fi} \mid \vec{x} := N(\vec{x}) \mid \mathsf{Com}\ ;\ \mathsf{Com}$$

That is, a command is either a skip command, an assignment, a conditional, a procedure call, or a sequence of other commands.

The semantics of this language is defined in terms of program states that the commands manipulate. A program state, σ, is a map from variables to values: $\sigma : x \to v$. Fig. 4 formalizes the semantics of commands by relating states before and after executing the command. A skip command leaves the state unchanged (E-SKIP). An assignment updates a program variable x to an given value e (E-ASSIGN). A conditional first evaluates the condition expression e, and if the result of evaluation result is the symbol True, then its evaluation is based on the first command (E-IFTRUE); otherwise, the evaluation is based on the second command (E-IFFALSE).

In order to support procedures, we also define a space of lookup tables **LTable**. A lookup table maps the names of procedures to a tuple: the parameters $\vec{p}$, the body of the procedure Com, and the output variables $\vec{o}$.

$$\mathbf{LTable} : N \to (\vec{p}, \mathsf{Com}, \vec{o})$$

A procedure call applies a lookup table $T \in \mathbf{LTable}$ to a procedure name N, constructs a program state σ' by substituting the parameters $\vec{i}$ by the arguments $\vec{x}$, evaluates the body over σ' to get σ'', and finally substitutes the return variables $\vec{r}$ by the value of the output variables $\vec{o}$ in σ''.

A program is a command paired with a lookup table.

3.2 Constrained Horn Clauses

PEQUOD finds valid program invariants using an external Constrained Horn Clause (CHC) solver. CHCs are a class of constraint programming problems [10]. Each CHC has the following form: $\texttt{chc} := \texttt{head} \Leftarrow \texttt{body}$. The clause head is an uninterpreted predicate applied to a set of variables. The clause body is the conjunction of a logical formula together with any number of uninterpreted predicates applied to variables. A CHC system is a set of CHCs together with a distinguished uninterpreted predicate, called the query.

A *solution* of a CHC system is a map from uninterpreted predicates to interpretations. A *valid* solution is one where **(1)** the interpretation of the query is the constant function that returns the proposition False and **(2)** replacing each uninterpreted predicate by its interpretation (instantiated with the applied variables) results in a set of valid implications.

4 Technical Approach

In this section, we describe our technical approach in detail. In §4.1, we define the proof system that PEQUOD uses to prove relational properties In §4.2, we describe a procedure for automatically finding proofs in this system.

4.1 Proof System

We define a proof system that extends standard Hoare Logic with new rules that can verify relational properties. To present this system, we first define the concept of a product command: $\mathsf{PCom} ::= \mathsf{Com} \times \mathsf{Com}$. Informally, product commands are used to represent pairs of independent program fragments whose execution we wish to consider simultaneously. This intuition can be formalized by the following semantic rule:

$$\text{E-PROD}\ \frac{\langle c_0, \sigma\rangle \Downarrow \sigma' \qquad \sigma \cap \tau = \varnothing \qquad \langle c_1, \tau\rangle \Downarrow \tau' \qquad \sigma' \cap \tau' = \varnothing}{\langle c_0 \times c_1, \sigma \cup \tau\rangle \Downarrow \sigma' \cup \tau'}$$

Because members of a product command share no vocabulary, we can reorder the members at will without changing the semantic meaning: The product of two commands is commutative with respect to the program semantics. Additionally, we enrich the vocabulary of commands with one additional constructor: $[\![\mathsf{Com}]\!]$. This command essentially adds a wrapper around the inner command. The semantic meaning of the wrapped command is the same as the inner command; we merely add this construction for the purposes of the proof rules.

Given these additional constructions, our proof system extends the standard Hoare Logic naturally such that it respects products. A relational invariant, P, is a first-order logical proposition that contains the vocabulary of each program. Judgments in the proof system take the following form:

$$T, \Gamma \vdash \{P\}\ \mathsf{PCom}\ \{Q\}$$

P and Q are relational invariants, where P is the precondition and Q is the postcondition. T is a lookup table that contains mappings from procedure names to procedure bodies (implemented as commands). Γ

$$\textsc{Skip}\frac{}{T,\Gamma \vdash \{P\}\ \mathsf{skip} \times \mathsf{skip}\ \{P\}} \qquad \textsc{Assign}\frac{T,\Gamma \vdash \{P\}\ \mathsf{skip} \times c\ \{Q\}}{T,\Gamma \vdash \{P[x \mapsto e]\}\ x := e \times c\ \{Q\}}$$

$$\textsc{If}\frac{\begin{array}{c}T,\Gamma \vdash \{P \wedge\ \ e\}\ c_0 \times c_2\ \{Q\} \\ T,\Gamma \vdash \{P \wedge \neg e\}\ c_1 \times c_2\ \{Q\}\end{array}}{T,\Gamma \vdash \{P\}\ \mathsf{if}\ e\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1\ \mathsf{fi} \times c_2\ \{Q\}} \qquad \textsc{Assume}\frac{\{P\}\ c_0 \times c_1\ \{Q\} \in \Gamma}{T,\Gamma \vdash \{P\}\ c_0 \times c_1\ \{Q\}}$$

$$\textsc{Step}\frac{T,\Gamma \cup \{\{P\}\ [\![c_0]\!] \times c_1\ \{Q\}\} \vdash \{P\}\ c_0 \times c_1\ \{Q\}}{T,\Gamma \vdash \{P\}\ [\![c_0]\!] \times c_1\ \{Q\}}$$

$$\textsc{Call}\frac{T[N] = (\vec{p}, c_0, \vec{o}) \qquad T,\Gamma \vdash \{P\}\ [\![c_0]\!] \times c\ \{R\}}{T,\Gamma \vdash \{P[\vec{p} \mapsto \vec{e}] \wedge \forall X'.(R[X \mapsto X'] \implies Q[\vec{x} \mapsto \vec{o}]\}\ \vec{x} := N(\vec{e}) \times c\ \{Q\}}$$

$$\textsc{Part}\frac{\begin{array}{cc}(c_0', c_0'') = \mathsf{part}(c_0) & (c_1', c_1'') = \mathsf{part}(c_1) \\ T,\Gamma \vdash \{P\}\ c_0' \times c_1'\ \{Q\} & T,\Gamma \vdash \{Q\}\ c_0'' \times c_1''\ \{R\}\end{array}}{T,\Gamma \vdash \{P\}\ c_0 \times c_1\ \{R\}}$$

$$\textsc{Cons}\frac{\begin{array}{c}T,\Gamma \vdash \{P'\}\ c_0 \times c_1\ \{Q'\} \\ P \Rightarrow P' \qquad Q' \Rightarrow Q\end{array}}{T,\Gamma \vdash \{P\}\ c_0 \times c_1\ \{Q\}} \qquad \textsc{Comm}\frac{T,\Gamma \vdash \{P\}\ c_1 \times c_0\ \{Q\}}{T,\Gamma \vdash \{P\}\ c_0 \times c_1\ \{Q\}}$$

Figure 5: Proof judgments for determining the validity of invariants over product command.

is a context, which is a set of Hoare Triples of the form $\{P\}\ \mathsf{PCom}\ \{Q\}$. Γ is used as a set of hypotheses which can be used to complete proofs of programs with recursion.

Fig. 5 presents the proof rules. Rules SKIP and ASSIGN simply model the semantics associated with the respective command in the context of a product. IF models the semantics of a conditional command. A critical difference between this rule and the rule from standard constructions is that the conditional is part of a product command. This allows the prover to reason about both branches of a conditional simultaneously with another program.

CALL models the semantics of a procedure call. Suppose that the prover wishes to prove an assertion over a pair of two commands: a call command $x := N(e)$ with an arbitrary command c. We use N for the name of the called procedure, c_0 for the body of the procedure, $\vec{p}$ for the vector of parameters, and $\vec{o}$ for the vector of return variables. If the prover demonstrates that the *wrapped* command $[\![c_0]\!]$ paired with c satisfies pre-condition P and post-condition R, then the call to N paired with c satisfies Q, given that the pre-condition $P[\vec{p} \mapsto \vec{e}]$ holds under an additional assumption. In Fig. 5, the assumption $\forall X'.(R[X \mapsto X'] \implies Q[\vec{x} \mapsto \vec{o}]$ means that R, which holds at the end of the called procedure, entails the post-condition Q in callee procedure, after substituting the output variables $\vec{x}$ by the return variables $\vec{o}$. X' a copy of variables in callee that are different from the output variables.

STEP is used to step into the body of a procedure. It allows the prover to add the current goal as a hypothesis. The rule unwraps the command while adding the goal to the hypothesis. Later, the prover can use ASSUME, which states that a proof goal can be satisfied if the goal is a hypothesis in the context Γ.

CONS is a typical component of a Hoare Logic system. It states that we can weaken the pre-condition

and strengthen the post-condition. As stated earlier, the semantics of commands are commutative with respect to products. COMM allows the prover to continue the proof by applying ASSIGN, IF, and CALL on either member of the product. This rule is critical for relational reasoning. In practice, the prover uses COMM to select the order in which to model the subcommands. The example in §2.1 shows that choosing the right order to apply COMM results in simple invariants that satisfy the proof.

Recall that a focus of our approach is finding appropriate synchronization points of the program in conjunction with relational invariants. PART is responsible for this reasoning. PART makes use of a procedure part. Informally, part allows the prover to partition a sequence of commands into two subsequences by cutting a sequence at an arbitrary point. PART decomposes a product command into two product commands, and proves them sequentially. The formal definition of part is given in the extended paper.

One key observation of this proof system is that this system is non-deterministic. In particular, PART allows the prover to subdivide the input programs at will: By choosing different partitionings, the prover is selecting synchronization points. COMM rule is also non-deterministic, and can be applied anywhere in the proof. Once a suitable ordering has been chosen by applying COMM and PART, the prover can potentially construct simple invariants that lead to a valid proof. Hence, the difficulty of designing the automatic proof system is determining how to use COMM and PART.

4.2 Verifying Relational Properties Automatically

Verifying a relational property of a product command, *pcd*, is reducible to deriving a relational Hoare Triple $\{P\}\ pcd\ \{Q\}$ under a given the lookup table T and an empty context Γ. The relational property is modeled by the pre-condition P and the post-condition Q. For example, the Hoare Triple given in §2 describes a property that specifies `tri0` and `tri1` are equivalent.

PEQUOD attempts to construct a proof of a relational Hoare Triple by iteratively executing three steps: First, PEQUOD constructs a bounded product command pcd' from the original product command *pcd* that respects a given bounding number n. pcd' is an under-approximation of *pcd* where each recursive procedure executes at most n times. In §4.2.1, we describe how to construct pcd' from *pcd* and n.

Second, PEQUOD generates a set of proofs for pcd' in a corresponding Hoare Triple. Because pcd' is bounded, PEQUOD can attempt all proof paths by exhaustively applying COMM and PART. PEQUOD populates these proofs with appropriate intermediate invariants using a system of Constrained Horn Clauses. In §4.2.2, we describe how to generate a set of proofs for a bounded Hoare Triple. In the third step, PEQUOD attempts to generalize the work done in the second step by finding a proof for the unbounded commands among the proofs for the bounded commands. By searching through the set of bounded proofs, PEQUOD is searching for synchronization points of the input programs that lead to a proof. In §4.2.3, we describe this generalization step in detail.

If PEQUOD cannot find a generalizable proof, then it increases n and starts again from the first step.

4.2.1 Constructing Bounded Programs

In order to represent bounded versions of programs, we extend our imperative command inductive definition with one additional constructor, $\bot$. $\bot$ should be understood as a command that immediately terminates. We use this construction to replace recursive calls to procedures outside the bound that we currently consider.

Bound is a procedure that constructs a bounded command c' and corresponding lookup table T' from an input command c with lookup table T and a bounding number n. The output command c' is

input : A Hoare Triple $\{P\}\ pcd'\ \{Q\}$ where $pcd' \in \mathsf{PCom}$ is bounded product command and its lookup table T.
output : A CHC system whose solution is a set of proofs for the given Hoare Triple.

```
1  Procedure ConstructCHC({P} pcd' {Q}, T)
2      CHC ← ∅
3      Procedure ConstructAux({P} pcd' {Q})
4          switch pcd' do
5              case skip × skip do
6                  return
7              otherwise do
8                  foreach c0 × c1 ∈ Permute(pcd') do
9                      switch c0 do
10                         case ⊥ do
11                             CHC ← CHC ∪ {Q ⇐ False}
12                         case c0'; c0'' do
13                             foreach (pcd0, pcd1) ∈ Partition(pcd') do
14                                 R ← freshRel
15                                 ConstructAux({P} pcd0 {R})
16                                 ConstructAux({R} pcd1 {Q})
17                         case x := e do
18                             R ← freshRel
19                             CHC ← CHC ∪ {[x ↦ e]R ⇐ P}
20                             ConstructAux({R} pcd' {Q})
21                         ...
22     ConstructAux({P} pcd' {Q})
23     return CHC
```

Algorithm 1: Given a Hoare Triple over a product command pcd' and a corresponding lookup table T, generate a CHC system that represents all possible proof paths for this triple.

an under-approximation of the input command c that respects n. The result of calling Bound is a new, bounded program where each recursive procedure is "copied" at most n times. Further calls to recursive procedures are modeled by $\bot$. In §2, Fig. 2 shows a bounded command $\texttt{tri0}_0$ with its lookup table that constructs from original command `tri0` in Fig. 1 with the bounded number three. The missing 'else' clauses in these examples correspond to the command $\bot$. An implementation of Bound is described in the extended paper.

4.2.2 Verifying Bounded Programs via Constrained Horn Clauses

PEQUOD constructs invariants for all possible proofs of a bounded Hoare Triple using a system of Constrained Horn Clauses (CHCs). Alg. 1 describes ConstructCHC, a procedure that constructs a CHC system representing all possible proofs for a given Hoare Triple. The solution of a CHC system is a set of relational invariants that support the set of proofs. If PEQUOD cannot find a solution of the constructed CHC system, then either **(1)** PEQUOD will find a counter-example of the relational property or **(2)** the underlying theorem prover does not support expressive enough logic to construct valid invariants.

ConstructCHC defines an auxiliary procedure ConstructAux. ConstructAux is a recursive descent

over the structure of the product command that accumulates a CHC system in the variable *CHC*. If pcd' is exactly the product command $\mathsf{skip} \times \mathsf{skip}$, then the recursion is finished and *CHC* contains a complete system. Otherwise, ConstructAux applies the procedure Permute on the product command pcd'. $\mathsf{Permute}(pcd')$ returns two product commands by applying the proof rule COMM. For each $c_0 \times c_1$ in set $\mathsf{Permute}(pcd')$, ConstructAux examines the first product c_0.

If c is a sequence of commands, then ConstructAux applies Partition to $c \times c_1$. Partition is a procedure that generates a set of all possible partitions of pcd'. $\mathsf{Partition}(c_0)$ and $\mathsf{Partition}(c_1)$ are the sets containing all valid partitions that respect the partition rule without duplicating skip. For each pair (pcd_0, pcd_1) in the set of partitions, ConstructAux constructs a fresh relational predicate R as an intermediate proposition. It then recurses on both parts. If c_0 is neither a sequence of commands nor $\bot$, then ConstructAux updates the CHC system based on the semantics of c_0 and recurses on pcd'. For example, if c_0 is an assignment then a clause is added which indicates the precondition implies the intermediate proposition with the appropriate substitution.

The key intuition behind ConstructCHC is that it constructs CHC system that contains all possible proofs for the bounded command pcd' by exhaustively applying the COMM and PART rules. When the constructed CHC system is solved, the solution contains invariants for all possible proofs of $\{P\}\ pcd'\ \{Q\}$. In practice, ConstructCHC includes optimizations that avoid redundant work.

PEQUOD solves CHC systems generated by ConstructCHC using an external solver. The solution σ is map from each relational predicate to its interpretation. Replacing each relational predicate by the corresponding invariant in the proof leads to valid Hoare Triples for the bounded program.

4.2.3 Generalizing Bounded Proofs

PEQUOD defines a procedure Syn that searches the set of proofs for the bounded commands to find one proof that can be generalized for the unbounded original commands. Syn operates over a bounded product command pcd' and corresponding looks up table T', as well as a solution σ of the constructed CHC system, which contains proper invariants for all proof paths. Syn decides if one generalizable proof path can be found for the original, unbounded command *pcd*, within the current set of bounded proof paths and its invariants.

The key intuition behind this algorithm is that Syn only needs to find **one** generalizable proof path among the set of bounded proof paths with current synthesized invariants. Syn has a similar structure to ConstructCHC with two key differences.

First, Syn attempts to use ASSUME to generalize the current proofs for the unbounded, original programs. ASSUME can only be applied when the context contains an appropriate Hoare Triple as hypothesis. Syn builds up context at each call site. When it revisits an identical command a second time (called procedure names can be different copies of the same original procedure), Syn checks if the pre-condition of the hypothesis is implied by the goal pre-condition and if the post-condition of the hypothesis implies the goal post-condition. If so, the hypothesis can be used to apply ASSUME to find a generalized proof for the current goal. For example, in §2.1, Fig. 3 depicts two bounded product commands $\texttt{tri0}_0 \times \texttt{tri1Aux}_0$ and $\texttt{tri0}_1 \times \texttt{tri1Aux}_1$. Both of these commands represent the same unbounded product command, $\texttt{tri0} \times \texttt{tri1Aux}$. Since the relational invariants for these two Hoare Triples are the same, Syn can use the first triple as an assumption to prove the second.

Second, Syn only needs to find one valid proof for the goal. Thus Syn can choose between all permutations/partitions of the bounded command pcd' to find one generalized proof for the original command *pcd*. Fig. 3 shows all possible proof paths for one sub-proof goal, and Syn only needs to find one proof path that passes through the node $\texttt{tri0}_1 \times \texttt{tri1Aux}_1$ to finish the proof. Other proof paths can

be discarded. The algorithm is presented more carefully in the extended paper.

5 Evaluation

We performed an empirical evaluation of PEQUOD to answer the following questions: How effective is PEQUOD compared to other automated relational verifiers?

To answer the above experimental questions, we implemented PEQUOD as a verifier of relational properties of programs represented in JVM bytecode. The only requirement imposed by PEQUOD on the logic for expressing program semantics is that the logic **(1)** has an effective decision procedure, which PEQUOD uses to check possible entailments (§4.2.2), and **(2)** can be encoded in the logic of constraints supported by its CHC solver. A subset of the JVM semantics can be encoded in the logic of linear arithmetic with arrays. This logic is supported both by the Z3 decision procedure and the DUALITY CHC solver implemented in Z3 [24].

We applied PEQUOD to benchmarks introduced in previous work on relational verification by *automatic induction* [23], programs and properties corresponding to theorems over recursive functions posed as theorem-proving exercises [20], and solutions to coding problems on the Leetcode platform [16]. We also slightly modified two benchmarks (`plusNSm0` and `sumSumAcc0`) that required a verifier to prove a corollary that is strictly weaker than key inductive mutual summary of the programs. Such modified benchmarks can present distinct challenges to a verifier because they require the verifier to synthesize non-trivial inductive summaries. The benchmarks require proofs of properties including equivalence, distributivity, monotonicity, commutativity, associativity, injectivity, transitivity, and symmetry.

We compared PEQUOD to implementations of techniques that perform automatic induction [23], that transform CHC systems encoding relational properties (VeriMapRel) [12], that use Cartesian Hoare Logic (CHL) [21], and that use self-composition. The current implementation of CHL does not support recursive procedures and self-composition cannot solve any but the simplest problem, `addDigits`. VeriMapRel does not support the negation of equality statements in its property specification, so we have to manually transformed the benchmarks with equality statements to a set of relational properties that use inequalities. Without this manual work, VeriMapRel can only solve two benchmarks. As a result, we have reported comparisons with automatic induction, and with VeriMapRel using this manual transformation.

Fig. 6 contains the results of our evaluation. In short, our experiments indicate that PEQUOD can efficiently verify properties beyond the scope of existing techniques. In particular, PEQUOD successfully verifies all but four of the benchmarks on which it was evaluated. Automatic induction fails to prove 15 cases within time that PEQUOD can. These cases require synthesizing non-trivial inductive relational invariants other than the given relational properteis to finish the proof. VeriMapRel fails to prove 8 cases within time that PEQUOD can. These cases requires sophisticated synchronization between two programs.

PEQUOD failed to converge on four cases because DUALITY did not generate relational invariants of bounded programs that can be generalized. This is a known challenge for CHC solvers that use an interpolating theorem prover [2]. For example, to prove that multiplication is commutative (**multComm**), PEQUOD requires the CHC solver to generate summaries that establish equalities over program variables, such as $\mathtt{x}_0 = \mathtt{y}_1$. Instead, the solver sometimes generates invariants specific to the structure of the hierarchical programs, such as $\mathtt{x}_0 = 1 \wedge \mathtt{y}_1 = 1$. However, because PEQUOD uses a CHC solver as a black box, it is well positioned to benefit directly from improvements to CHC solvers. Furthermore, in a significant number of cases, PEQUOD synthesize proper synchronization points with relational invariants from DUALITY's solutions that could not be found by existing techniques. The current implementation of Pequod

Source	Name	Property	Time(s)	Mem(MB)	[23]	[12]
	multMultAcc	equiv	5.6	183.0	✓	✓
	multMultAcc0	equiv	5.5	182.8	✓!	✓
	multL1	equiv	4.7	185.8	✓	✗
	multR1	equiv	2.7	121.8	✓	✓
	multDistL	distr	12.6	381.3	✗	✗
Automating	**multDistR**	distr	23.9	433.6	✓	✗
Induction	**sumSimple**	equiv	2.8	121.2	✓	✓
For Solving	**sumDown**	equiv	5.2	197.1	✓	✓
Horn Clauses	**sumUp**	equiv	5.0	177.3	✓	✓
[23]	**sumUpDown**	equiv	6.1	179.2	✗	✗
	sumSumAcc	equiv	5.5	179.7	✓	✓
	sumSumAcc0‡	equiv	11.7	253.6	✗	✓
	multAssoc	assoc	33.3	599.4	✗	✗
	sumMono	mono	TO	1530.6	✓	✓
	multMono	mono	TO	1368.3	✓!	✓
	multComm	comm	TO	2028.8	✓	✓
	plusComm	comm	6.0	254.2	✗	✓
	plusAssoc	assoc	38.9	618.0	✗	✗
	plusNSm	equiv	21.0	430.6	✓	✓
	plusNSm0‡	equiv	22.9	434.7	✗	✓
Software	**plusRearrange**	equiv	138.5	658.6	✗	✓
Foundations	**doublePlus**	equiv	4.3	118.2	✗	✓
	doubleInjective	inj	4.8	237.5	✗	✓
	evenbS	equiv	26.6	642.2	✗	✗
	beqNatSym	sym	5.4	202.3	✓	✓
	beqNatTrans	tran	7.2	242.4	✓	✓
	mult0plus	equiv	TO	786.1	✗	✗
	addDigits†	equiv	2.5	67.2	✗	✓
	trailingZeroes	equiv	5.1	200.7	✗	✓
LeetCode	**climbStairs**†	equiv	6.3	258.0	✗	✗

Figure 6: The results of our evaluation of PEQUOD. Each benchmark is labeled with its source, name, the class of relational property that PEQUOD attempted to verify, time spent by PEQUOD to synthesize a proof, the peak amount of memory that PEQUOD used, and whether *automated induction* [23] or *VeriMapRel* [12] verified the benchmark. A time of *TO* denotes that PEQUOD was unable to converge within 300 seconds. The superscript '!' denotes that automated induction only converged with a manually-provided lemma. Each benchmark with the superscript '‡' is a minor modification of the original benchmark immediately above it. The superscript '†' denotes that the benchmarks obtained from the source were not equivalent. In such cases, the data reports the performance of PEQUOD when applied to a version of the benchmark that we manually patched to be correct.

and executable benchmarks are available online.[1]

6 Related Work

Previous work [12, 18, 23] has established that problems in relational verification can be reduced to solving systems of Constrained Horn Clauses, and has proposed novel proof systems for solving such systems. Such systems are expressive, and can be partially automated. However, they require a prover to manually provide lemmas that the system establishes by induction when a lemma stronger than the goal invariant must be proved [23] (analogous to suggesting inductive invariants when they must differ from the goal invariant of a program) or direct how relational predicates in a given system should be paired in order to generate a solvable system [12, 18]. PEQUOD performs such reasoning automatically.

Previous work has proposed frameworks that allow a prover to verify that recursive programs satisfy a mutual summary [3, 4, 11, 14, 15], but require the user to direct how procedures must be paired, and in some cases provide mutual summaries. Other approaches for verifying relational properties of single-procedure programs have been significantly automated [21], but the developed automation tactics are carefully tuned to syntactic forms of the programs and would be non-trivial to generalize to programs that contain multiple procedures.

Verifying relational properties can also be reduced to synthesizing a suitable *product program* [5, 7]. Some approaches synthesize product programs in the class of *sequential compositions* automatically, but such product compositions either cannot easily be constructed manually [9] or can only prove relational properties in a heavily restricted class [5, 8, 13, 17, 22]. Other approaches construct product programs depending partly on matching control structures between the pairs of programs and establishing the logical equivalence of program conditions included in matched structures. Previous work has also explored constructing *asymmetric product programs* [6] which can express proofs of relational properties not provable in the system used by PEQUOD. However, such work does not address the problem of automatically inferring loop invariants of the synthesized product program, which may be viewed alternatively as the mutual summary between loops of the original programs.

Recent work has introduced logics for reasoning about relational properties of higher-order programs [1]. However, these systems have not yet been used to automatically synthesize proofs of program equivalence. PEQUOD can only synthesize proofs for first-order recursive programs, but can do so automatically.

References

[1] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg & Pierre-Yves Strub (2017): *A Relational Logic for Higher-Order Programs*. In: *ICFP*, doi:10.1145/3110265.

[2] Aws Albarghouthi & Kenneth L. McMillan (2013): *Beautiful Interpolants*. In: *CAV*, doi:10.1007/978-3-642-39799-8_22.

[3] David A. Naumann Anindya Banerjee & Mohammad Nikouei (2016): *Relational Logic with Framing and Hypotheses*. In: *FSTTCS*, doi:10.4230/LIPIcs.FSTTCS.2016.11.

[4] John D. Backes, Suzette Person, Neha Rungta & Oksana Tkachuk (2013): *Regression Verification Using Impact Summaries*. In: *SPIN*, doi:10.1007/978-3-642-39176-7_7.

[5] Gilles Barthe, Juan Manuel Crespo & César Kunz (2011): *Relational Verification Using Product Programs*. In: *FM*, doi:10.1007/978-3-642-21437-0_17.

[1] https://www.dropbox.com/s/yks0eyic8dsf69e/pequod.zip?dl=0

[6] Gilles Barthe, Juan Manuel Crespo & César Kunz (2013): *Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification*. In: *LNCS*, doi:10.1007/978-3-642-35722-0_3.

[7] Gilles Barthe, Juan Manuel Crespo & César Kunz (2016): *Product Programs and Relational Program Logics*. In: *JLAMP*, doi:10.1016/j.jlamp.2016.05.004.

[8] Gilles Barthe, Pedro R. D'Argenio & Tamara Rezk (2004): *Secure Information Flow by Self-Composition*. In: *CSFW-17*, doi:10.1017/S0960129511000193.

[9] Lennart Beringer (2011): *Relational Decomposition*. In: *ITP*, doi:10.1007/978-3-642-22863-6_6.

[10] Nikolaj Bjørner, Kenneth L. McMillan & Andrey Rybalchenko (2013): *On Solving Universally Quantified Horn Clauses*. In: *SAS*, doi:10.1007/978-3-642-38856-9_8.

[11] Marcel Böhme, Bruno C. d. S. Oliveira & Abhik Roychoudhury (2013): *Partition-based regression verification*. In: *ICSE*, doi:10.1109/ICSE.2013.6606576.

[12] Alberto Pettorossi Emanuele De Angelis, Fabio Fioravanti & Maurizio Proietti (2016): *Verifying Relational Program Properties by Transforming Constrained Horn Clauses*. In: *CILC*.

[13] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer & Mattias Ulbrich (2014): *Automating regression verification*. In: *ASE*, doi:10.1145/2642937.2642987.

[14] Benny Godlin & Ofer Strichman (2009): *Regression verification*. In: *DAC*, doi:10.1145/1629911.1630034.

[15] Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri & Henrique Rebêlo (2013): *Towards Modularly Comparing Programs Using Automated Theorem Provers*. In: *CADE-24*, doi:10.1007/978-3-642-38574-2_20.

[16] (2016): *LeetCode Online Judge*. `https://leetcode.com/`. Accessed: 2015 Nov 16.

[17] Nuno P. Lopes & José Monteiro (2016): *Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic*. *STTT* 18(4), doi:10.1007/s10009-015-0366-1.

[18] Mattias Ulbrich Moritz Kiefer, Vladimir Klevanov (2016): *Relational Program Reasoning Using Compiler IR*. In: *VSTTE*, doi:10.1007/s10817-017-9433-5.

[19] Lauren Pick, Grigory Fedyukovich & Aartic Gupta (2018): *Exploiting Synchrony and Symmetry in Relational Verification*. In: *CAV*, doi:10.1007/978-3-319-96145-3_9.

[20] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg & Brent Yorgey (2018): *Logical Foundations*. Software Foundations series, volume 1, Electronic textbook. Version 5.5. `http://www.cis.upenn.edu/~bcpierce/sf`.

[21] Marcelo Sousa & Isil Dillig (2016): *Cartesian Hoare logic for verifying k-safety properties*. In: *PLDI*, doi:10.1145/2980983.2908092.

[22] Tachio Terauchi & Alexander Aiken (2005): *Secure Information Flow as a Safety Problem*. In: *SAS*, doi:10.1007/11547662_24.

[23] Hiroshi Unno & Sho Torii (2017): *Automating Induction for Solving Horn Clauses*. In: *CAV*, doi:10.1007/978-3-319-63390-9_30.

[24] (2017): *Z3Prover/z3 - GitHub*. `https://github.com/Z3Prover/z3`. Accessed: 2017 July 1.