Relational Verification using Reinforcement Learning

JIA CHEN, University of Texas at Austin, USA
JIAYI WEI, University of Texas at Austin, USA
YU FENG, University of California, Santa Barbara, USA
OSBERT BASTANI, University of Pennsylvania, USA
ISIL DILLIG, University of Texas at Austin, USA

Relational verification aims to prove properties that relate a pair of programs or two different runs of the same program. While relational properties (e.g., equivalence, non-interference) can be verified by reducing them to standard safety, there are typically *many* possible reduction strategies, only some of which result in successful automated verification. Motivated by this problem, we propose a new relational verification algorithm that learns useful reduction strategies using reinforcement learning. Specifically, we show how to formulate relational verification as a Markov decision process (MDP) and use reinforcement learning to synthesize an optimal policy for the underlying MDP. The learned policy is then used to guide the search for a successful verification strategy. We have implemented this approach in a tool called Coeus and evaluate it on two benchmark suites. Our evaluation shows that Coeus solves significantly more problems within a given time limit compared to multiple baselines, including two state-of-the-art relational verification tools.

CCS Concepts: • **Software and its engineering → Software verification**; • **Theory of computation →** Reinforcement learning.

Additional Key Words and Phrases: verification, relational property, reinforcement learning, policy gradient, neural network, proof search

ACM Reference Format:
Jia Chen, Jiayi Wei, Yu Feng, Osbert Bastani, and Isil Dillig. 2019. Relational Verification using Reinforcement Learning. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 141 (October 2019), 30 pages. https://doi.org/10.1145/3360567

1 INTRODUCTION

Relational verification aims to establish that two programs—or a pair of executions of a program—do not interact in unintended ways. Such *relational properties* appear under many guises when reasoning about program correctness. For instance, a prototypical relational property is *program equivalence* which requires that two programs have the same observable behavior when executed on the same input. Other examples include *non-interference* [Goguen and Meseguer 1982], which

Authors' addresses: Jia Chen, Department of Computer Science, University of Texas at Austin, Austin, Texas, 78712-0233, USA, jchen@cs.utexas.edu; Jiayi Wei, Department of Computer Science, University of Texas at Austin, Austin, Texas, 78712-0233, USA, jiayi@cs.utexas.edu; Yu Feng, Department of Computer Science, University of California, Santa Barbara, Santa Barbara, California, 93106-5110, USA, yufeng@cs.ucsb.edu; Osbert Bastani, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania, 19104-6309, USA, obastani@seas.upenn.edu; Isil Dillig, Department of Computer Science, University of Texas at Austin, Austin, Texas, 78712-0233, USA, isil@cs.utexas.edu.

2475-1421/2019/10-ART141
https://doi.org/10.1145/3360567

is used for reasoning about side channels, as well as algebraic properties like *injectivity* and *anti-symmetry* [Sousa and Dillig 2016]. In addition, relational properties also arise in the context of software evolution [Lahiri et al. 2012, 2013] and version control [Sousa et al. 2018].

Due to their significance across many application domains, relational properties have been the subject of much attention in the program verification literature [Barthe et al. 2011, 2016, 2012; Benton 2004; De Angelis et al. 2016b; Mordvinov and Fedyukovich 2017; Sousa and Dillig 2016; Yang 2007]. Interestingly, a common theme among all relational verification techniques is that they reduce the problem of proving a relational property to that of standard safety. For instance, given two programs P_1 and P_2 that need to obey some relational property, a popular approach is to construct a so-called *product program* P such that the relational property is valid if P obeys some safety property [Barthe et al. 2011, 2016; Eilers et al. 2018; Sousa et al. 2018; Zaks and Pnueli 2008]. In a similar vein, many relational program logics [Benton 2004; Chen et al. 2017; Sousa and Dillig 2016; Yang 2007] reduce relational verification to the problem of discharging a set of standard Hoare triples.

Despite the power and conceptual simplicity of this approach, a key challenge in relational verification is that there are typically *many* ways to reduce the original problem to safety. While each reduction method corresponds to a valid proof strategy, some of these strategies are much more amenable to automation than others. For example, consider verifying equivalence between programs P_1, P_2 in Figure 1 and the two product programs A, B shown in Figure 2. Here, both A and B have the property of being safe if and only if P_1 and P_2 are equivalent. However, it is significantly easier for most automated tools to prove the assertion in B (see comments in Figure 2).

In principle, there is a simple way to deal with this challenge: We could simply try *all* possible ways of reducing the relational verification problem to standard safety and conclude that the property holds if *any* of the corresponding safety problems can be verified. Unfortunately, this naïve strategy is not feasible in practice because there are simply too many reduction strategies to try. As a result, prior techniques either require the user to manually specify a suitable reduction strategy (e.g., [Barthe et al. 2011; Felsing et al. 2014]) or use domain-specific heuristics (e.g., [Chen et al. 2017; Sousa and Dillig 2016; Sousa et al. 2018; Zaks and Pnueli 2008]). The former strategy is sub-optimal in that it lacks automation, whereas the latter approach is time-consuming and highly domain-dependent. In particular, developing good hand-crafted heuristics for relational verification requires both domain expertise and knowledge about the underlying safety checker.

This paper aims to address this challenge by guiding relational proof search using machine learning. That is, given a benchmark suite of relational verification tasks from a specific domain and an underlying safety verifier, our goal is to automatically learn a probability distribution over possible reduction strategies such that those deemed more promising by the machine learning model are explored first. This approach allows our relational verification algorithm to automatically infer useful search heuristics for new problems without requiring costly user intervention.

However, one key challenge to using machine learning in this context is the lack of *labeled* training data in the form of successful relational proof strategies. In this paper, we address this challenge by using *reinforcement learning (RL)*, which effectively allows the relational verifier to learn over time from its *own* failed and successful proof attempts. Specifically, in an offline phase, we use RL to train the relational verifier on a corpus of verification problems such that the verifier is "rewarded" for using reduction strategies that result in successful proofs. Then, in an online phase, the verifier leverages the knowledge accumulated in the offline training phase to solve new verification problems much more efficiently.

One of the key contributions of this paper is to show how to formalize the relational verification problem as a *Markov decision process (MDP)*. In our formulation, states of the MDP correspond to partial proofs, and a *policy* of the MDP specifies which reduction strategy to use at each proof step.

```
int P1(int *a) {
  int max = a[0], i;
  for (i = 1; i < N; ++i)
    if (a[i] > max)
      max = a[i];
  return max;
}
```

```
int P2(int *a) {
  int max, i;
  for (i = 0; i < N; ++i)
    if (i == 0) max = a[i];
    else if (a[i] > max)
      max = a[i];
  return max;
}
```

Fig. 1. Example programs

```
void A(int *a0, int *a1) {
  assume(a0 == a1);
  int max0 = a0[0], max1, i0, i1;
  for (i0 = 1, i1 = 0; i0 < N || i1 < N; ++i0, ++i1) {
    /* We need quantified loop invariant to state that:
     * - max0 is the largest element in a0[0..i0]
     * - max1 is the largest element in a1[0..i1] */
    if (i0 < N)
      if (a0[i0] > max0) max0 = a0[i0];
    if (i1 < N)
      if (i1 == 0) max1 = a1[i1];
      else if (a1[i1] > max1) max1 = a1[i1];
  }
  assert(max0 == max1);
}
```

```
void B(int *a0, int *a1) {
  assume(a0 == a1);
  int max0 = a0[0], max1, i0, i1;
  i1 = 0;
  if (i1 == 0) max1 = a1[i1]; else /* Not relevant */;
  for (i0 = 1, i1 = 1; i0 < N && i1 < N; ++i0, ++i1) {
    /* Loop invariant: i0 == i1 && max0 == max1 */
    if (a0[i0] > max0) max0 = a0[i0];
    if (i1 == 0) /* Not relevant */;
    else if (a1[i1] > max1) max1 = a1[i1];
  }
  assert(max0 == max1);
}
```

Fig. 2. Product programs for Figure 1. Program A requires a complex quantified loop invariant, whereas B can be verified using the simple loop invariant $i_0 = i_1 \wedge max_0 = max_1$.

Given this formulation, we give a technique for finding an optimal policy of the MDP by adapting the *policy gradient algorithm* to solve the optimization problem that arises in the off-line phase of our algorithm. Then, in the on-line phase, we use a backtracking search algorithm that leverages the optimal policy learned during the off-line phase to efficiently search through different strategies for reducing the relational verification problem to standard safety.

We have implemented the proposed relational verification approach in a tool called Coeus and evaluate it on two benchmarks suites. In our first experiment, we use Coeus to validate the correctness of source-to-source transformations performed by the ROSE compiler infrastructure from the Lawrence Livermore Laboratory. In our second experiment, we use Coeus to prove relational properties between programs written by different people, such as different solutions to programming challenge problems. Our evaluation shows that the proposed approach solves significantly more benchmarks compared to multiple baselines, including two state-of-the-art verification tools. In particular, among a total of 259 relational verification benchmarks that we use in our evaluation, Coeus can successfully verify 88% of the problems whereas existing state-of-the-art verifiers solve less than half.

To summarize, this paper makes the following key contributions:

- We propose a new relational verification algorithm that performs proof search using a policy that is obtained using reinforcement learning.
- We show how to formulate the relational verification problem as a Markov decision process, and we propose a variant of the policy gradient technique to find an optimal policy for the corresponding MDP.
- We describe a backtracking search algorithm that uses the learned policy to guide proof search.
- We experimentally evaluate our approach on two benchmark suites and empirically quantify the benefits of our approach over competing techniques.

$$\frac{\vdash \{\Phi\} S \{\Psi\}}{\vdash \langle \Phi \rangle skip \circledast S \langle \Psi \rangle} \text{ (Lift)}$$

$$\frac{\vdash \{\Phi\} S \{\Phi'\} \qquad \vdash \langle \Phi' \rangle S_1 \circledast S_2 \langle \Psi \rangle}{\vdash \langle \Phi \rangle S; S_1 \circledast S_2 \langle \Psi \rangle} \text{ (Seq)}$$

$$\frac{\begin{array}{c}\Phi \Rightarrow (e_1 \leftrightarrow e_2) \qquad \Phi \Rightarrow \mathbb{I} \\ \vdash \langle \mathbb{I} \wedge e_1 \wedge e_2 \rangle S_1 \circledast S_2 \langle \mathbb{I} \rangle \\ \vdash \langle \mathbb{I} \wedge \neg e_1 \wedge \neg e_2 \rangle S \circledast S' \langle \Psi \rangle\end{array}}{\vdash \langle \Phi \rangle \textbf{while } e_1 \textbf{ do } S_1; S \circledast \textbf{while } e_2 \textbf{ do } S_2; S' \langle \Psi \rangle} \text{ (Sync)}$$

$$\frac{\begin{array}{c}\vdash \langle \Phi \wedge e \rangle S; \textbf{while } e \textbf{ do } S; S_1 \circledast S_2 \langle \Psi \rangle \\ \vdash \langle \Phi \wedge \neg e \rangle S_1 \circledast S_2 \langle \Psi \rangle\end{array}}{\vdash \langle \Phi \rangle \textbf{while } e \textbf{ do } S; S_1 \circledast S_2 \langle \Psi \rangle} \text{ (Peel)}$$

$$\frac{\begin{array}{c}f_1 = \lambda \vec{p_1}.\, S_1' \qquad f_2 = \lambda \vec{p_2}.\, S_2' \qquad \vdash \langle \oplus' \rangle S_1' \circledast S_2' \langle \ominus' \rangle \\ \Phi \Rightarrow \oplus'[\vec{a_1}/\vec{p_1}, \vec{a_2}/\vec{p_2}] \\ \vdash \langle \ominus'[\vec{a_1}/\vec{p_1}, \vec{a_2}/\vec{p_2}] \rangle S_1 \circledast S_2 \langle \Psi \rangle\end{array}}{\vdash \langle \Phi \rangle \textbf{call } f_1(\vec{a_1}); S_1 \circledast \textbf{call } f_2(\vec{a_2}); S_2 \langle \Psi \rangle} \text{ (Call)}$$

Fig. 3. Selected rules for reducing 2-safety verification problem to standard Hoare triples

Organization. This paper is organized as follows: In Sections 2 and 3, we provide some background on relational verification and discuss how we represent proof strategies for relational verification problems. Next, in Section 4, we give a high-level overview of our approach and motivate our design choices. Section 5 introduces our learning objective and explains how to solve this optimization problem using reinforcement learning. Then, Section 6 presents our policy-guided proof search algorithm, and Sections 7 and 8 discuss our implementation and experimental results. Finally, we discuss related work in Section 9.

2 BACKGROUND ON RELATIONAL VERIFICATION

As mentioned in Section 1, existing techniques reduce relational verification to safety checking either by explicitly constructing a product program [Barthe et al. 2011, 2016; Eilers et al. 2018] or introducing a proof system where certain proof obligations can be discharged by an off-the-shelf safety checker [Barthe et al. 2012; Benton 2004; Sousa and Dillig 2016]. In this paper, we adopt the latter approach and think of relational verification as the problem of searching for a proof within a relational program logic.

Following prior work [Benton 2004; Chen et al. 2017; Sousa and Dillig 2016], we assume a relational program logic that derives *relational Hoare triples* of the form $\langle \Phi \rangle\ S_1 \circledast S_2\ \langle \Psi \rangle$ where S_1, S_2 are programs over disjoint sets of variables and Φ (resp. Ψ) is a relational precondition (resp. post-condition). For example, for equivalence checking, Φ would stipulate that the inputs of the two programs are equal, and Ψ would assert that the outputs are equal.

By studying prior work on relational program verification [Barthe et al. 2011, 2016; Benton 2004; Felsing et al. 2014; Sousa and Dillig 2016], we built a library of 37 different proof rules and tactics, of which five representative ones are shown in Figure 3. While a detailed discussion of these proof rules is out of scope for this paper, we highlight some of their salient features below.

Reduction to safety. As illustrated by the Lift and Seq rules from Figure 3, the premises of a relational proof rule can involve proving standard Hoare triples of the form $\{P\}S\{Q\}$. Thus, relational program logics eventually reduce the problem to standard safety checking.

Non-determinism. Given a proof goal $\mathcal{G} = \langle\Phi\rangle\ S_1 \circledast S_2\ \langle\Psi\rangle$, there are typically many rules that can be used to prove $\mathcal{G}$. For example, if S_1 and S_2 are both while loops, we can apply three different rules (namely, Seq, Sync, and Peel) even for the small subset of proof rules shown in Figure 3.

Sensitivity to proof strategy. Let us define a *proof strategy* to be a mapping from each proof subgoal to a proof rule that can be used for discharging it. Because the base cases of a relational proof require invoking an off-the-shelf safety checker, the success of a particular proof strategy depends on how easy or difficult the corresponding safety checking problems are. Thus, some proof strategies may lead to successful proofs, while others may not.

Large search space. Since there are many proof rules that can be used to discharge a relational Hoare triple, the search space of proof strategies is very large. Specifically, given m rules with k subgoals and two programs S_1, S_2 of size n, the size of the search space is $\Theta((mk)^n)$. Thus, in practice, it is often infeasible to explore all possible proof strategies within a reasonable time limit.

Shape of the rules. As we can see from Figure 3, each relational proof rule $\mathcal{R}$ consists of (i) a goal $\mathcal{G}$ (i.e., a relational Hoare triple), (ii) a set of subgoals $\Omega = \{\mathcal{G}_1, \ldots, \mathcal{G}_n\}$, where each $\mathcal{G}_i$ is also a relational Hoare triple, and (iii) a set of verification conditions (VCs) (e.g., $\Phi \rightarrow (e_1 \leftrightarrow e_2)$ and $\Phi \rightarrow \mathbb{I}$ in rule Sync). Thus, we can represent each relational proof rule $\mathcal{R}$ as a quadruple $\mathcal{R} = (\mathcal{R}_{id}, \mathcal{R}_{\mathcal{G}}, \mathcal{R}_{\Omega}, \mathcal{R}_{\varphi})$, where $\mathcal{R}_{id}$ is the name of the rule, $\mathcal{R}_{\mathcal{G}}, \mathcal{R}_{\Omega}$ represent the goal and subgoals respectively, and $\mathcal{R}_{\varphi}$ is a formula that corresponds to the conjunction of all VCs. Observe that the VCs can involve unknown predicates such as $\mathbb{I}$ in rule Sync or pre- and post-conditions $\mathcal{P}, \mathcal{Q}$ in rule Call; thus we represent VCs as a system of *Constrained Horn Clauses (CHCs)* [De Angelis et al. 2016b; Mordvinov and Fedyukovich 2017]. Furthermore, since standard Hoare triples can also be encoded as CHCs [Bjørner et al. 2015; De Angelis et al. 2016a], we also think of the standard Hoare triples that occur in the premises as part of the VC of the corresponding rule.

3 REPRESENTING PROOF STRATEGIES

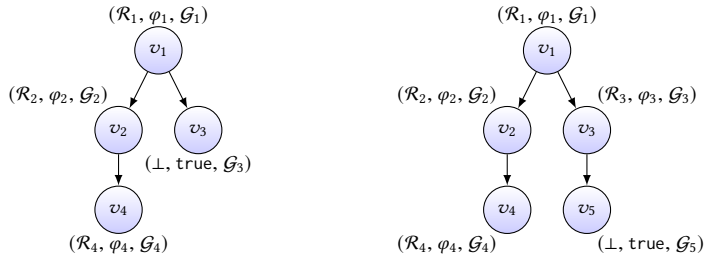
Our goal in the rest of the paper is to automate relational verification by efficiently searching through a large space of possible proof strategies. In this section, we describe our representation of proof strategies and formalize what we mean by a strategy being successful.

Intuitively, a proof strategy specifies which rule to apply to discharge each subgoal. In this paper, we represent relational proof strategies as trees where nodes correspond to proof subgoals and edges represent the application of some proof rule.

*Definition 3.1 (***Proof strategy).* A *proof strategy* is a tuple $\Upsilon = (V, E, A_{\mathcal{R}}, A_{\varphi}, A_{\mathcal{G}})$ where

- V is a set of nodes.
- E is a set of arcs.
- $A_{\mathcal{R}}$ maps each node to either a proof rule $\mathcal{R}$ or $\perp$.
- A_{φ} maps each node to a verification condition.
- $A_{\mathcal{G}}$ maps each node to the corresponding proof goal $\langle\Phi\rangle\ S_1 \circledast S_2\ \langle\Psi\rangle$ for its subtree.

We refer to $A_{\mathcal{R}}, A_{\varphi}, and A_{\mathcal{G}}$ as the *rule*, *VC*, and *goal* annotations respectively, and we use the symbol $\perp$ to indicate *open branches* of the proof. That is, if $A_{\mathcal{R}}(v)$ is $\perp$ where $v \in V$, this means that we have not yet chosen a proof rule for proving the subgoal associated with v. Thus, we also differentiate between *complete* and *incomplete* proof strategies:



(a) Before applying $\mathcal{R}_3$ (b) After applying $\mathcal{R}_3$

Fig. 4. Example proof strategies

*Definition 3.2 (***Complete proof strategy***)*. We say that $v \in V$ is an *open branch* of a relational proof strategy if $A_{\mathcal{R}}(v) = \bot$. A proof strategy is *complete* if it does not have any open branches and *incomplete* otherwise.

Example 3.3. Figure 4a shows an example proof strategy Υ. Based on the tree structure, we see that nodes v_2 and v_3 correspond to subgoals of v_1, which represents the proof goal $\mathcal{G}_1$. Furthermore, since v_1 is annotated with rule $\mathcal{R}_1$, we can tell that proof subgoals v_2 and v_3 were obtained by applying proof rule $\mathcal{R}_1$. Also, node v_1 is annotated with verification condition φ_1; this means φ_1 must be discharged for the application of rule $\mathcal{R}_1$ to be valid. Finally, note that v_3 is an open branch of the proof since we have $A_{\mathcal{R}}(v_3) = \bot$. Thus, Υ is *incomplete*.

Since our verification algorithm starts with a completely unconstrained strategy and iteratively refines it, we define the notion of *initial strategy* for a given proof goal $\mathcal{G}$:

*Definition 3.4 (***Initial strategy***)*. Given a relational proof goal $\mathcal{G}$, the *initial strategy* for $\mathcal{G}$, denoted $\Upsilon_0(\mathcal{G})$, is given by:

$$(\{v_1\}, \emptyset, \{v_1 \mapsto \bot\}, \{v_1 \mapsto \mathit{true}\}, \{v_1 \mapsto \mathcal{G}\})$$

Thus, $\Upsilon_0(\mathcal{G})$ encodes all possible ways of proving goal $\mathcal{G}$ within the given relational program logic. Since our verification algorithm will iteratively refine its strategy by expanding an open branch, Algorithm 1 describes how we apply a proof rule $\mathcal{R}$ to strategy Υ. Given an incomplete strategy Υ and a proof rule $\mathcal{R}$, ApplyProofRule yields a refined strategy by generating (a) verification conditions as prescribed by $\mathcal{R}_\varphi$, and (b) new proof subgoals $\mathcal{G}_1, \ldots, \mathcal{G}_n$ according to $\mathcal{R}_\mathcal{G}$. Note that base cases in the proof system do not generate subgoals, and introduction of subgoals results in the addition of new open branches in the refined strategy. [1]

Example 3.5. Figure 4b shows the result of applying rule $\mathcal{R}_3$ to the open branch of Figure 4a. Here, $\mathcal{R}_3$ generates one new subgoal $\mathcal{G}_5$ with associated verification conditions φ_3. The rule application introduces a new open branch v_5 below v_3, with $A_\varphi(v_5)$ initialized to `true`.

*Definition 3.6 (***Strategy refinement***)*. We say that a strategy Υ' *directly refines* another strategy Υ, written $\Upsilon' \preceq^1 \Upsilon$, if Υ' is the result of calling ApplyProofRule on Υ for *some* proof rule $\mathcal{R}$. We define $\preceq$ as the reflexive transitive closure of $\preceq^1$ and say that Υ' *refines* Υ whenever $\Upsilon' \preceq \Upsilon$.

Given a proof strategy, we need a way of determining whether it results in a valid proof. Towards this goal, we define a *successful* proof strategy as follows:

[1] In Algorithm 1, `GenVC` and `GenSubgoals` take a proof rule and a proof goal as input and generate new VCs and new subgoals according to the proof rules in Figure 3, respectively. The `FirstOpenBranch` function returns the first open branch of the given strategy. Since every open branch must be closed eventually, we assume a canonical order for simplicity.

Algorithm 1 Rule application

Input: $\Upsilon = (V, E, A_{\mathcal{R}}, A_{\varphi}, A_{\mathcal{G}})$: incomplete proof strategy
Input: $\mathcal{R} = (\mathcal{R}_{id}, \mathcal{R}_{\mathcal{G}}, \mathcal{R}_{\Omega}, \mathcal{R}_{\varphi})$: rule to apply
Output: A refined proof strategy

procedure ApplyProofRule(Υ, $\mathcal{R}$)
  $v \leftarrow$ `FirstOpenBranch`(Υ)
  $A_{\mathcal{R}}(v) \leftarrow \mathcal{R}_{id}$
  $A_{\varphi}(v) \leftarrow$ `GenVC`($\mathcal{R}_{\varphi}, A_{\mathcal{G}}(v)$)
  $\Omega \leftarrow$ `GenSubgoals`($\mathcal{R}_{\Omega}, A_{\mathcal{G}}(v)$)
  for $\mathcal{G}_i \in \Omega$ **do**
    $v' \leftarrow$ fresh node
    $(V,\ E) \leftarrow (V \cup \{\ v'\ \},\ E \cup \{\ v \rightarrow v'\ \})$
    $(A_{\mathcal{R}},\ A_{\varphi}) \leftarrow (A_{\mathcal{R}}[v' \leftarrow \bot],\ A_{\varphi}[v' \leftarrow \mathit{true}])$
    $A_{\mathcal{G}} \leftarrow A_{\mathcal{G}}[v' \leftarrow \mathcal{G}_i]$
  return $(V, E, A_{\mathcal{R}}, A_{\varphi}, A_{\mathcal{G}})$

*Definition 3.7 (***Successful strategy***).* A proof strategy $\Upsilon = (V, E, A_{\mathcal{R}}, A_{\varphi}, A_{\mathcal{G}})$ is *successful* if

- Υ is complete.
- The formula $\bigwedge_{v \in V} A_{\varphi}(v)$ can be proven satisfiable.

Recall from Section 2 that we represent verification conditions as Constrained Horn Clauses (CHCs) in this paper. Thus, the satisfiability of the formula $\bigwedge_{v \in V} A_{\varphi}(v)$ means that there exists an interpretation of the unknown relations under which the formula evaluates to true.

*Definition 3.8 (***Failing proof strategy***).* A proof strategy $\Upsilon = (V, E, A_{\mathcal{R}}, A_{\varphi}, A_{\mathcal{G}})$ is failing if the conjunction $\bigwedge_{v \in V} A_{\varphi}(v)$ is unsatisfiable.

Note that, unlike successful proof strategies, failing strategies need not be complete. In particular, the formula can become unsatisfiable $\bigwedge_{v \in V} A_{\varphi}(v)$ even when the proof contains open branches. Our proof search algorithm will take advantage of this observation in Section 6.

4 OVERVIEW

In this section, we give a high-level overview of our relational verification algorithm and highlight its salient features.

Searching for relational proofs. As mentioned earlier, our verification algorithm performs backtracking search over proof strategies, prioritizing those that are most promising. To this end, we use reinforcement learning to predict which proof strategies are most likely to be successful. Specifically, our reinforcement learning algorithm produces a distribution p over complete proof strategies such that, if $p(\Upsilon_1) > p(\Upsilon_2)$, then Υ_1 is more likely to be a successful strategy compared to Υ_2 according to the learned model.

Given a specific relational verification task t, we use the notation $p^{(t)}$ to denote the distribution of complete proof strategies Υ that are *applicable* to verifying t (i.e., the root node of Υ is annotated with the initial proof goal for t). Now, to solve a relational verification problem t, our search algorithm initializes $p_0 = p^{(t)}$. Then, on each iteration $i = 0, 1, 2, \ldots$ (up to some upper bound r)[2], it chooses a complete proof strategy Υ_i that has high probability according to p_i, and checks whether

[2]While the value of r used by the search algorithm is large (it corresponds to the timeout set on the search algorithm), during training we choose r to be small. By doing so, we encourage the search algorithm to discover a successful proof strategy earlier in the search.

Υ_i is successful. If so, the verification algorithm terminates and returns Υ_i. Otherwise, based on feedback explaining why Υ_i was unsuccessful, our algorithm constrains the support of p_i to obtain a new distribution p_{i+1} that avoids making mistakes similar to those in Υ_i. In Section 6, we describe how our search strategy constructs p_{i+1} given p_i and a failing proof strategy Υ_i.

Learning objective. The goal of our learning algorithm is to generate a distribution p that places high probability mass on successful proof strategies. In particular, it aims to solve the following optimization problem:

$$p^* = \arg\max_p \Pr_{t\sim\mathcal{T},\Upsilon\sim\xi^{(t)}_{r,p}}[O(\Upsilon) = 1] \tag{1}$$

Here, $t \sim \mathcal{T}$ is a uniformly random task, $O(\Upsilon)$ is 1 if Υ is successful and 0 otherwise, and $\xi^{(t)}_{r,p}$ is a distribution of proof strategies explored by the search algorithm, i.e.,

$$\xi^{(t)}_{r,p}(\Upsilon) = \frac{1}{r}\sum_{i=1}^{r} p_i^{(t)}(\Upsilon).$$

Essentially, the objective in Eq. (1) is to maximize the probability that our search algorithm discovers a successful proof strategy for a uniformly random task within r iterations.

However, there are three challenges to solving the optimization problem from Eq. (1): First, we do not have positive examples of successful proof strategies. Second, we only have a finite training set of tasks $\mathcal{T}_{\text{train}}$. Finally, standard reinforcement learning algorithms cannot be applied to optimize Eq. (1) due to the modified distribution $\xi^{(t)}_{r,p}$. Below, we discuss how we address these challenges.

Reinforcement learning. Since we do not have positive examples of successful proof strategies, we cannot use standard supervised learning algorithms to optimize Eq. (1). Instead, we have oracle access to O in the form of our proof checker, which makes it possible to use reinforcement learning. In Section 5.2, we describe how to formulate the optimization problem from Eq. (1) as a reinforcement learning problem.

Function approximation. Since we are only given a finite subset of tasks $\mathcal{T}_{\text{train}} \subseteq \mathcal{T}$, we can only approximate the samples $t \sim \mathcal{T}$ from Eq. (1) with uniformly random samples $t \sim \mathcal{T}_{\text{train}}$. However, the solution to the approximate objective may not generalize to all of $\mathcal{T}$. Thus, we use a *feature map* to improve generalization. The essential idea is to restrict the search space to distributions $p(\Upsilon)$ that only depend on Υ through a handcrafted feature map $\phi(\Upsilon) \in \mathcal{X} = \mathbb{R}^d$, which is designed to map similar proof strategies to similar features. In particular, given two strategies Υ and Υ', we should have $\phi(\Upsilon) \approx \phi(\Upsilon')$ if the proof goals labeling their roots are similar, and $\phi(\Upsilon) \not\approx \phi(\Upsilon')$ otherwise. Then, if the optimal distribution p^* assigns high probability mass to Υ, it similarly assigns high probability mass to Υ' (assuming p^* is reasonably smooth). Thus, knowledge can be transferred to new tasks with proof goals that are different from those for training tasks $t \in \mathcal{T}$. We describe this approach in Section 5.3.

Reinforcement learning algorithm. Standard reinforcement learning algorithms can only be applied to optimizing Eq. (1) for the case $\xi^{(t)}_{r,p} = p^{(t)}$, i.e., where $r = 1$. In other words, these algorithms can only optimize for the case where the search algorithm only considers a single proof strategy, so they are not directly applicable to our setting where the search algorithm tries multiple consecutive proof strategies.

One straightforward idea to solve this problem is to extend the horizon of the learning algorithm and encode a history of every proof step taken so far. While such a solution would allow us to account for past proof attempts, it suffers from two problems (namely state space explosion and delayed reward) that make training prohibitively slow. Thus, rather than using this naive strategy,

we propose an alternative solution for better solving the optimization problem from Eq. 2. We describe this adaptation in Section 5.4.

5 REINFORCEMENT LEARNING

We now describe how to use reinforcement learning to generate a distribution p over complete proof strategies that is used to guide our search algorithm. We first start with a brief primer on Markov decision processes and reinforcement learning and then explain their application to our relational verification problem.

5.1 Background on Reinforcement Learning

A reinforcement learning problem is typically specified as a *Markov decision process (MDP)*. Informally, an MDP is a transition system where the process is in some state S_i at each time step, and a decision maker can take any of the actions $A_1, \ldots, A_n$ that is available at state S_i and collects some reward R. The goal of reinforcement learning is to find the optimal action to take in each state to maximize the expected long-term reward.

Definition 5.1. A *Markov decision process* is a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{S}_0, \mathcal{S}_F, \mathcal{A}, \mathcal{P}, \mathfrak{R})$, where $\mathcal{S}$ is the set of states, $\mathcal{S}_0$ is the initial distribution over states, $\mathcal{S}_F$ is a set of terminal states, $\mathcal{A}$ is the set of actions, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the (possibly stochastic) transition function, and $\mathfrak{R} : \mathcal{S} \to \mathbb{R}$ is the (possibly stochastic) reward function.[3]

Definition 5.2. A *policy* π for an MDP $\mathcal{M}$ is a (possibly stochastic) function $\pi : \mathcal{S} \to \mathcal{A}$ specifying which action to take in each state.

We can use π to select which action to take at each state, which results in a (random) trajectory through the state space. This trajectory is referred to as a *rollout*:

Definition 5.3. A *rollout* $\zeta \sim \pi$ is a random sequence of tuples $\zeta \in (\mathcal{S} \times (\mathcal{A} \cup \{\varnothing\}) \times \mathfrak{R})^*$ constructed as follows:

- sample a random state $S_0 \sim \mathcal{S}_0$
- sample actions $A_i = \pi(S_i)$, random transitions $S_{i+1} = \mathcal{P}(S_i, A_i)$, and rewards $R_i = \mathfrak{R}(S_i)$ for each $i \in \{1, \ldots, T\}$ until a terminal state $S_T \in \mathcal{S}_F$ is reached.

Then, ζ is the sequence

$$((S_0, A_0, R_0), \ldots, (S_{T-1}, A_{T-1}, R_{T-1}), (S_T, \varnothing, R_T)).$$

Note that there is no action A_T for the last tuple since S_T is a terminal state.

As mentioned earlier, the goal in reinforcement learning is to maximize expected long-term reward:

Definition 5.4. Given an MDP $\mathcal{M}$, the *reinforcement learning problem* is to find the *optimal policy* $\pi^* = \arg\max_\pi \mathfrak{R}^{(\pi)}$, where $\mathfrak{R}^{(\pi)}$ denotes the *cumulative reward* of π:

$$\mathfrak{R}^{(\pi)} = \mathbb{E}_{\zeta \sim \pi}\left[\sum_{i=0}^{T} R_i\right].$$

Example 5.5. Figure 5 (a) shows an example of a robot planning task where the goal is to find the treasure. The MDP representing this task is shown in Figure 5 (b). The states $\mathcal{S}$ are the circles, the transitions $\mathcal{P}$ are the edges, the actions $\mathcal{A} = \{\text{right}, \text{down}\}$ are labels on the edges, and the

[3] Oftentimes, a *discount factor* $\gamma \in (0, 1)$ is needed to ensure that the learning problem for the MDP is well-defined; however, in our setting, the MDP always terminates after a finite number of steps.

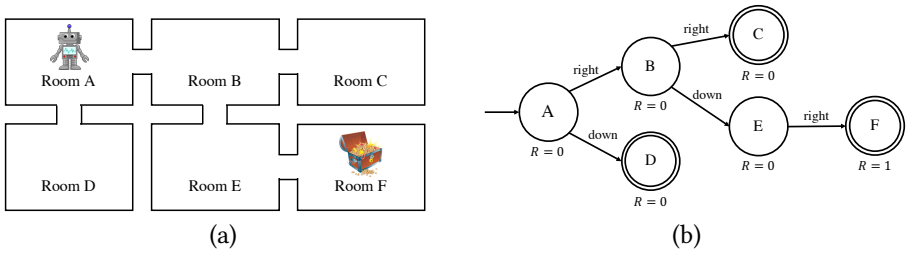


Fig. 5. (a) Example of a simple planning task, where the goal of the robot is to find the treasure. (b) Representation of the task as a Markov decision process (MDP).

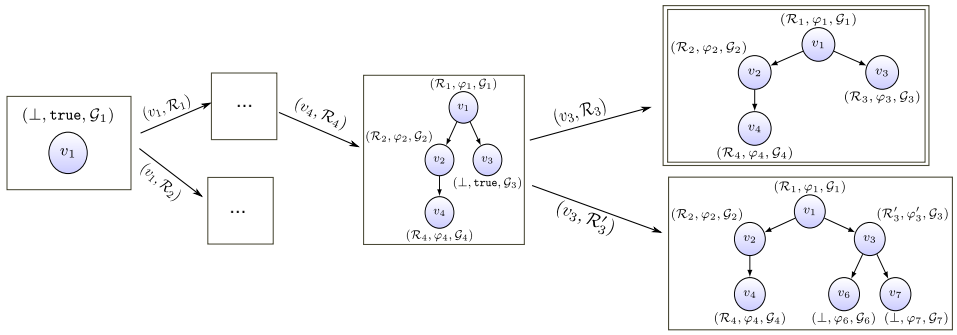


Fig. 6. An example of an MDP constructed for a relational verification problem.

rewards $\Re$ are shown below the states. [4] The initial state set $\mathcal{S}_0$ contains just A (i.e., $S_0 = A$ with probability 1), and the final states are $\mathcal{S}_F = \{C, D, F\}$. The following are two examples of policies for this MDP: [5]

$$\pi_1(A) = \pi_1(B) = \pi_1(E) = \text{right}$$
$$\pi_2(A) = \pi_2(E) = \text{right}, \quad \pi_2(B) = \text{down}.$$

Then, the cumulative rewards are $R^{(\pi_1)} = 0$ (since this policy terminates in state C) and $R^{(\pi_2)} = 1$ (since this policy terminates in state F).

5.2 MDP for Relational Verification

To use reinforcement learning in our setting, we need to formulate an MDP $\mathcal{M}_{\text{proof}}$ encoding relational verification problems. Intuitively, given an (incomplete) proof strategy Υ, we want to learn a policy that chooses a proof rule $\mathcal{R}$ to apply to Υ that maximizes the chance of eventually constructing a successful proof strategy. Thus, states in our MDP are proof strategies, and actions are proof rules that can be applied to the current strategy. We begin by describing the MDP constructed for a single task. Then, we describe how to construct an MDP that encodes a distribution of tasks $\mathcal{T}$.

Single-task MDP. Given a single relational verification task expressed as a proof goal $\mathcal{G}$, we construct the MDP $\mathcal{M}_{\text{proof}}(\mathcal{G}) = (\mathcal{S}, \mathcal{S}_0, \mathcal{S}_F, \mathcal{A}, \mathcal{P}, \Re)$ as follows:

- The states $\mathcal{S}$ are proof strategies Υ.

[4] If an action A is *unavailable* in a state S (i.e., there is no edge coming out of S with label A), then it is treated as a self-loop.
[5] We can omit defining the policy for states C, D, and F since there are no available actions in these states.

- $\mathcal{S}_0$ corresponds to the initial proof strategy $\Upsilon_0(\mathcal{G})$ for $\mathcal{G}$ (recall Def. 3.4); i.e., the initial state is $\Upsilon_0(\mathcal{G})$ with probability 1.
- The terminal states $\mathcal{S}_F$ are complete proof strategies.
- The actions $A \in \mathcal{A}$ are all pairs $(v, \mathcal{R})$, where $\mathcal{R}$ is a proof rule that can be applied to node v in the current proof strategy Υ.
- The (deterministic) transitions are $\mathcal{P}(S, A) = S'$, where S' is the proof strategy obtained from S by applying the proof rule A to the first open branch of S.
- The reward function is $\mathfrak{R}(S) = \mathcal{O}(S)$ (i.e., the reward is 1 if S is successful and 0 otherwise).

Intuitively, the actions in $\mathcal{M}_{\text{proof}}$ incrementally construct a complete proof strategy $S_T \in \mathcal{S}_F$ from the initial proof strategy S_0, and the reward is whether S_T is successful.

Example 5.6. Figure 6 shows an example of an MDP for a relational verification problem $\mathcal{G}_1$. Each state is a proof strategy Υ, and each action is a pair $(v, \mathcal{R})$ consisting of a node v in the current proof strategy and a proof rule $\mathcal{R}$ that can be applied to v. The initial state is the left-most state. For each action, an arrow shows the state transition that would occur if that action is taken. The right-most state on the top is a final state with reward 1 since it represents a successful proof strategy; all other states have reward 0.

Task-distribution MDP. The MDP $\mathcal{M}_{\text{proof}}$ representing a *distribution* of relational verification tasks is exactly the same as the single-task MDP $\mathcal{M}_{\text{proof}}(\mathcal{G})$, except for the distribution $\mathcal{S}_0$ over initial states. In particular, a state $S_0 \in \mathcal{S}_0$ is sampled by sampling a task $t \sim \mathcal{T}$, and then letting S_0 be the initial proof strategy $\Upsilon_0(\mathcal{G}_t)$ for the goal $\mathcal{G}_t$ corresponding to t.

Connection to our objective. Now, we describe the connection between the optimal policy for $\mathcal{M}_{\text{proof}}$ and the optimization problem from Eq. (1). First, we define a correspondence between distributions p over complete proof strategies and MDP policies π:

Definition 5.7. Given a policy π for $\mathcal{M}_{\text{proof}}$, its *terminal state distribution* is

$$p_\pi(\Upsilon) = \Pr_{\zeta \sim \pi}(S_T = \Upsilon),$$

where S_T is the terminal state of rollout ζ.

In other words, $p_\pi(\Upsilon)$ is the probability that a rollout $\zeta \sim \pi$ ends in terminal state $S_T = \Upsilon$. Since the terminal states in $\mathcal{M}_{\text{proof}}$ are complete proof strategies, p_π is a distribution over complete proof strategies. Then, we have the following theorem, which relates the problem of maximizing Eq. (1) to the reinforcement learning problem for our MDP $\mathcal{M}_{\text{proof}}$ [6]:

Theorem 5.8. *Let π^* be the optimal policy for $\mathcal{M}_{proof}$, and*

$$p^* = \arg\max_p Pr_{t \sim \mathcal{T}, \Upsilon \sim p^{(t)}}[\mathcal{O}(\Upsilon) = 1], \tag{2}$$

where p is a distribution over complete proof strategies. Then, we have $p^* = p_{\pi^*}$.

There is a key difference between our objective Eq. (1) and the objective Eq. (2) from Theorem 5.8: In Eq. (1), the probability is taken with respect to complete proof strategies $\Upsilon \sim \xi_{r,p}^{(t)}$ (i.e., the distribution of proof strategies tried by our search algorithm given guiding distribution $p^{(t)}$), whereas in Eq. (2), the probability is taken with respect to $\Upsilon \sim p^{(t)}$ (i.e., a single proof strategy according to $p^{(t)}$). In other words, our objective optimizes over a sequence of complete proof strategies tried by the search algorithm, whereas Eq. (2) optimizes for a single randomly sampled proof strategy.

[6] Proofs of all theorems are in the Appendix.

The point of Theorem 5.8 is to show that existing reinforcement learning algorithms cannot be directly applied to optimizing Eq. (1). In particular, the optimal strategy computed by standard reinforcement learning maximizes the probability of finding a successful proof strategy in a *single* attempt, but we want to compute a policy that maximizes our chances of finding a successful proof during a conflict-driven search algorithm that explores many different relational proof strategies. In Section 5.4, we describe how we can adapt an existing reinforcement learning algorithm to optimize Eq. (1) instead of Eq. (2).

5.3 Function Approximation

Recall that, when we only have a limited set of training tasks available, then the solution to Eq. (1) may not generalize well beyond tasks in the training set. As standard, we use approximate reinforcement learning to improve generalization power [Sutton and Barto 2018]. We first give background on the approximate RL and then describe our design choices within this framework

Background on approximate reinforcement learning. In approximate reinforcement learning, one needs to provide:

- A *feature map* $\phi : \mathcal{S} \to \mathcal{X}$, where $\mathcal{X} = \mathbb{R}^d$, which maps each state S to a feature vector $\phi(S)$ representing S.
- A *function family* $f_\theta : \mathcal{X} \to \mathcal{A}$, parameterized by $\theta \in \Theta = \mathbb{R}^m$, which maps feature vectors to actions.

Then, rather than search over all possible policies, the reinforcement learning algorithm restricts to policies of the form $f_\theta(\phi(S))$ (for $\theta \in \Theta$). For example, in *deep reinforcement learning*, the function family f_θ takes the form of a deep neural network, where θ corresponds to the weights of the network.

Definition 5.9. Given a feature map $\phi : \mathcal{S} \to \mathcal{X}$ and function family f_θ, the *approximate reinforcement learning problem* is to compute the optimal parameters

$$\theta^* = \arg\max_{\theta \in \Theta} \mathfrak{R}^{(\theta)}, \tag{3}$$

where $\mathfrak{R}^{(\theta)} = \mathfrak{R}^{(\pi_\theta)}$ and $\pi_\theta(S) = f_\theta(\phi(S))$.

In other words, the goal of approximate reinforcement learning is to find a policy within function family f_θ that maximizes expected cumulative reward.

In order for approximate reinforcement learning to be effective, the feature map ϕ must be constructed using domain expertise to balance two competing goals: First, given two states S and S', if the most promising actions to take in S and S' are similar, then we should have $\phi(S) \approx \phi(S')$. On the other hand, if the most promising actions are very different, then we should have $\phi(S) \not\approx \phi(S')$. Thus, if the RL algorithm learns the best actions to take in state S, this knowledge is automatically transferred to taking good actions in state S' (assuming smoothness of f_θ).

Feature map. Since our proof strategies are complex tree-structured objects involving many relational Hoare triples, our feature map grossly over-approximates the states in $\mathcal{M}_{\text{proof}}$. Specifically, we design $\phi(\Upsilon)$ to take into account both (a) the global aspects of the proof tree (e.g., depth and breadth of its tree structure, number of open/closed branches, etc.) as well as (b) local properties of the first open branch of Υ. For (b), suppose that the active open branch is labeled with the proof goal $\mathcal{G} = \langle \Phi \rangle\ S_1 \circledast S_2\ \langle \Psi \rangle$. We featurize this relational Hoare triple by both considering which proof rules are (syntactically) applicable for discharging $\mathcal{G}$ and also performing a lightweight "diff" between S_1 and S_2. In particular, our differencing algorithm considers features such as whether both S_1, S_2 start with the same type of statement, whether they involve loops or recursive functions,

the ratio between their iteration count and step size (if both start with loops) etc. Thus, intuitively two strategies Υ_1 and Υ_2 will be deemed similar under ϕ if (a) their tree structures are similar, and (b) the same proof rule is likely to be successful for discharging the first open branches of Υ_1 and Υ_2.

Function family. In addition to the feature map, we also need a function family f_θ for mapping features (i.e., proof strategies) to actions (i.e., proof rules). For this, we use a standard choice in the reinforcement learning literature, namely the function family f_θ of neural networks with two (fully-connected) hidden layers and ReLU activations. Then, θ is the concatenation of all the weight and bias parameters of the layers in the neural network [Bastani et al. 2018a; Montgomery and Levine 2016; Schulman et al. 2015; Xiong et al. 2017].

Approximating our objective. Given the feature map and function family described above, we can approximate Eq. (1) as follows. First, given parameters $\theta \in \Theta$, we define its *terminal state distribution* to be $p_\theta = p_{\pi_\theta}$ (i.e., p_θ is a distribution over complete proof strategies defined by parameter θ). Then, rather than optimize over all distributions p, we restrict to optimizing over proof strategies of the form p_θ (for $\theta \in \Theta$):

$$\theta^* = \arg\max_{\theta\in\Theta} \Pr_{t\sim\mathcal{T}, S\sim\xi^{(t)}_{r,\theta}}[O(S) = 1], \tag{4}$$

where $\xi_{r,\theta} = \xi_{r,p_\theta}$. Observe that Eq. 4 differs from the standard approximate reinforcement learning problem in the same way Eq. 2 differs from Eq. 1: That is, rather than finding parameters of θ that maximize the likelihood of finding a successful proof in a *single* attempt, we want to find parameters that maximize our chances of finding a proof during a backtracking search algorithm.

5.4 Reinforcement Learning Algorithm

Recall from Section 5.2 that an optimal policy for our MDP does not yield an optimal solution to Eq. 1 (or Eq. 4 when we use approximation). In particular, standard RL algorithms maximize the expected cumulative reward under the assumption that we will explore a *single* rollout of the learned policy, whereas we want to maximize expected cumulative reward when exploring multiple rollouts during a backtracking search algorithm. Towards this goal, we describe a modified reinforcement learning algorithm that directly optimizes for our objective.

Our proposed optimization method builds on the *policy gradient algorithm*, which optimizes the cumulative reward $\mathfrak{R}^{(\theta)}$ as a function of the policy parameters $\theta \in \Theta$ using stochastic gradient descent. There are two key reasons for building on top of the policy gradient algorithm: First, as we discuss in the rest of this section, policy gradient is easy to adapt to directly optimize our objective. Second, because our feature vector $\phi(\Upsilon)$ grossly overapproximates Υ, we run into the so-called *perceptual aliasing* problem [Chrisman 1992; McCallum 1993], where two states that are different look the same under ϕ. In contrast to alternative algorithms like Q-learning, it is well-known that policy gradient works better in this scenario.

Background on policy gradient. The key challenge solved by the policy gradient algorithm is how to compute an estimate of the gradient $\frac{\mathrm{d}}{\mathrm{d}\theta}\mathfrak{R}^{(\theta)}$. This algorithm is based on the the following well-known *policy gradient theorem* [Sutton et al. 2000]:

Theorem 5.10. *We have*

$$\frac{\mathrm{d}}{\mathrm{d}\theta}\mathfrak{R}^{(\theta)} = \mathbb{E}_{\zeta\sim\pi_\theta}[\ell(\zeta)],$$

where

$$\ell(\zeta) = \sum_{i=0}^{T-1}\left(\sum_{j=i+1}^{T} R_j\right)\frac{\mathrm{d}}{\mathrm{d}\theta}\log\pi_\theta(S_i, A_i).$$

Intuitively, in Theorem 5.10, the term $\frac{d}{d\theta} \log \pi_\theta(S_i, A_i)$ gives a direction in the parameter space that, when moving the policy parameters towards it, increases the probability of taking action A_i at state S_i. Also note that the sum $\sum_{j=i+1}^{T} R_j$ is the total future reward after taking action A_i. In other words, $\ell(\zeta)$ is simply the sum of different directions in the parameter space weighted by their corresponding future reward. Thus, the gradient $\frac{d}{d\theta} R^\theta$ moves the policy parameters in a direction that increases the probability of taking actions associated with higher rewards.

Observe that Theorem 5.10 immediately gives a way to optimize the policy: Since we can compute the gradient of the objective $\Re^{(\theta)}$, we can use gradient descent to optimize $\Re^{(\theta)}$ as a function of the policy parameters θ.

Our algorithm. We now describe our algorithm for optimizing our objective in Eq. (4), i.e.,

$$J(\theta) = \Pr_{t \sim \mathcal{T}, S \sim \xi_{r,\theta}^{(t)}}[O(S) = 1].$$

To solve this problem, we leverage additional structure of our search algorithm: Recall that, given guiding distribution p over complete proof strategies, our search algorithm initializes $p_0 = p^{(t)}$, and then iteratively constructs a sequence of distributions $p_0, p_1, p_2, \ldots, p_r$. As we describe in Section 6, this sequence of distributions corresponds to a sequence of policies $\pi_{\theta,0}, \pi_{\theta,1}, \pi_{\theta,2}, \ldots, \pi_{\theta,r}$, where $p_i = p^{(t)}_{\pi_{\theta,i}}$. Then, we have the following theorem:

Theorem 5.11. *We have*

$$\frac{\mathrm{d}J}{\mathrm{d}\theta}(\theta) = \frac{1}{r} \sum_{i=1}^{r} \mathbb{E}_{\zeta \sim \pi_{\theta,i}}[\ell(\zeta)],$$

where $\ell(\zeta)$ is the same as in Theorem 5.10.

Intuitively, the key difference between our algorithm and standard policy gradient is that we maximize the likelihood that we will find a successful proof strategy during search rather than during a single rollout. In particular, the gradient of our modified reward is computed by sampling rollouts from r different distributions rather than a single distribution. Each distribution is obtained from the previous one by (a) sampling a rollout ζ_i from the current distribution p_i, and (b) if ζ_i corresponds to a failing proof strategy, inferring other failing strategies (see Section 6.2) to constrain the support of p_i.

As in standard policy gradient, we can use known techniques [Sutton et al. 2000] to approximate the gradient of $J(\theta)$:

$$\frac{\mathrm{d}J}{\mathrm{d}\theta}(\theta) \approx \frac{1}{r} \sum_{i=1}^{r} \frac{1}{n} \sum_{k=1}^{n} \ell(\zeta^{(i,k)}),$$

where $\zeta^{(i,k)} \sim \pi_{i,\theta}$. Thus, we can use this approximate gradient in conjunction with gradient descent to compute the optimal parameters:

$$\theta^* = \arg\max_{\theta \in \Theta} J(\theta).$$

6 POLICY-GUIDED PROOF SEARCH

In this section, we show how to use the optimal policy π synthesized using reinforcement learning to perform backtracking search over proof strategies.

Our relational verification algorithm, called RelVerif, is shown in Algorithm 2. Given a relational Hoare triple $\mathcal{G}$ and the stochastic policy π learned from the training examples, RelVerif returns a successful proof strategy if one exists and $\perp$ otherwise. At a high level, the algorithm works as follows: It maintains a worklist W of (incomplete) proof strategies, which initially contains

Algorithm 2 Policy-guided backtracking proof search

Input: $\mathcal{G}$ - target proof goal
Input: π - learned stochastic policy
Input: Δ - available proof rules
Output: A successful proof strategy for $\mathcal{G}$, or $\bot$ if it does not exist

```
1:  procedure RelVerif(𝒢, π, Δ)
2:      W ← {Υ₀(𝒢)}                                   ▷ worklist of proof strategies
3:      B ← ∅                                          ▷ blocked proof strategies
4:      while W ≠ ∅ do
5:          Υ ← ChooseStrategy(π, W)                   ▷ Use policy
6:          W ← W \ {Υ}
7:          for ℛᵢ ∈ Δ do
8:              if ¬Applies(ℛᵢ, Υ) then
9:                  continue
10:             Υᵢ ← ApplyProofRule(Υ, ℛᵢ)
11:             if ∃Υ′ ∈ B. Υᵢ ⪯ Υ′ then
12:                 continue
13:             if IsSuccessful(Υᵢ) then return Υᵢ
14:             if IsFailing(Υᵢ) then
15:                 B ← B ∪ { Minimize(Υᵢ) }
16:             else if ¬IsComplete(Υᵢ) then
17:                 W ← W ∪ { Υᵢ }
18:     return ⊥
```

the unconstrained strategy $\Upsilon_0(\mathcal{G})$ (recall Def. 3.4). During each iteration, the algorithm invokes a procedure called `ChooseStrategy`, discussed in Section 6.1, to pick the most promising strategy according to policy π (line 5) and constructs a series of refinements $\Upsilon_1, \ldots, \Upsilon_n$ by applying each one of the applicable proof rules $\mathcal{R}_i$ in the relational proof system Δ (line 10). If we are guaranteed that Υ_i is a failing strategy (i.e., Υ_i is a refinement of one of the *blocked strategies* B), then we move on to the next proof rule without adding Υ_i to the worklist W (lines 11-12). On the other hand, if Υ_i is successful (i.e., it is complete and the corresponding CHCs are satisfiable), then we return Υ_i as a solution to the relational verification problem (line 13). Otherwise, if Υ_i is failing, we compute an unsatisfiable core of the VCs used in Υ_i and add the corresponding *minimal failing strategy* to the blocked strategies B (lines 14-15). The use of blocking set B allows us to prune strategies that are guaranteed to be unsuccessful.

6.1 Using Policy to Guide Search

In order to use policy π to guide search, we need a suitable way to prioritize which states to explore first. Intuitively, we want our search algorithm to have two desired properties: First, complete proof strategies that have a higher probability of being successful according to p_π should be explored first. Second, to guarantee completeness of our approach, the search must be exhaustive. That is, given a large enough time limit, the algorithm should return a successful proof strategy if one exists.

One straightforward way to utilize π is to use a *stochastic* search algorithm that repeatedly samples complete proof strategies according to the distribution given by p_π. However, implementing an efficient random sampling algorithm that guarantees exhaustiveness is a challenging task. Instead, we use a *deterministic* search algorithm that simply enumerates complete proof strategies in decreasing order of their probability according p_π. The intuition is that strategies that are more

probable under p_π are more likely to lead to a successful proof; thus, they should be investigated first.

To ensure that the algorithm prioritizes complete strategies that correspond to more likely rollouts of π, we introduce a prioritization function ℓ_π as follows:

$$\ell_\pi(\Upsilon) = \begin{cases} 1 & \text{if } \Upsilon = \Upsilon_0(\mathcal{G}) \\ \ell_\pi(\Upsilon') - \log \Pr[\pi(\Upsilon') = \mathcal{R}] & \text{otherwise,} \end{cases}$$

where Υ =ApplyProofRule$(\Upsilon', \mathcal{R})$. Note that for a complete proof strategy Υ, we have $\ell_\pi(\Upsilon) = -\log p_\pi(\Upsilon)$. Thus, complete proof strategies that are more likely to be successful according to p_π are assigned a lower value according to ℓ_π.

Going back to Algorithm 2, the function `ChooseStrategy` simply uses the function ℓ_π to figure out which proof strategy to dequeue from W. In particular, `ChooseStrategy` dequeues the strategy with the lowest ℓ_π value.

Theorem 6.1. *Let Υ_1 and Υ_2 be two complete non-failing proof strategies. If $p_\pi(\Upsilon_1) > p_\pi(\Upsilon_2)$, then Υ_1 will be explored (i.e., dequeued from W) before Υ_2 by Algorithm 2.*

6.2 Finding Minimal Failing Strategies

To avoid exploring failing strategies that share the same *root cause* of failure as previously explored ones, our proof search algorithm uses *minimal failing proof strategies* to block strategies that are guaranteed to be unsuccessful. More, formally, a minimal failing proof strategy is defined as follows:

*Definition 6.2 (***Minimal failing proof strategy***).* Given a failing proof strategy Υ, we say that Υ' is a minimally failing proof strategy of Υ if the following conditions hold:

- $\Upsilon \preceq \Upsilon'$
- Υ' is failing
- There does not exist $\Upsilon'' \neq \Upsilon'$ such that $\Upsilon' \preceq \Upsilon''$.

Essentially, a minimally failing proof strategy Υ' for Υ captures the root cause of failure in the sense that every proof rule in Υ' is necessary for generating an unsatisfiable system of CHCs in Υ. Thus, any proof strategy that refines Υ' is also guaranteed to fail and can be pruned from the search space without losing completeness.

The `Minimize` procedure used at line 15 of Algorithm 2 computes a minimum failing strategy as follows: First, it computes a *minimal unsatisfiable core* of the VCs for a given failing strategy $\Upsilon = (V, E, A_\mathcal{G}, A_\mathcal{R}, A_\varphi)$. Then, it identifies a subset of nodes $V_\perp \subseteq V$ such that $\bigwedge_{v \in V_\perp} A_\varphi(v)$ is unsatisfiable but for every $U \subset V_\perp$ we have $\bigwedge_{v \in U} A_\varphi(v)$ is satisfiable. Hence, $V_\perp$ has the following key properties:

- If we remove nodes that are not in $V_\perp$ from Υ, we still get a failing strategy.
- Removing any node in $V_\perp$ from Υ will make it not failing.

In other words, we can view $V_\perp$ as the root cause of failure for strategy Υ; thus, all nodes that are descendants of $V_\perp$ can be removed from Υ while preserving unsatisfiability. The `Minimize` algorithm essentially removes all nodes $V_\perp$ from Υ but adds open branches as necessary to ensure that the resulting proof strategy is structurally well-formed.

The following theorem states that our search algorithm does not prune any successful proof strategies:

Theorem 6.3. *If there exists a complete proof strategy Υ for goal $\mathcal{G}$ such that $\bigwedge_{v \in V} A_\varphi(v)$ can be proven satisfiable by the CHC solver, then Algorithm 2 will produce a proof of correctness of $\mathcal{G}$.*

7 IMPLEMENTATION

We have implemented the proposed ideas in a prototype called Coeus. Our tool takes as input two C programs and a relational property and outputs a successful proof strategy if the property can be verified.

As depicted schematically in Figure 7, Coeus consists of three major components: First, the Proof System component implements the relational proof rules for reducing relational verification to standard safety. The Reinforcement Learning component implements the learning algorithm described in Section 5 and requires a set of representative training examples. Finally, the Proof Search component implements the backtracking search algorithm discussed in Section 6.

The Reinforcement Learning module is implemented in Python and uses the PyTorch library [Paszke et al. 2017]. The Proof System and the Proof Search components are both implemented in OCaml and use the front-end of the CompCert compiler [Leroy 2009] for parsing the input C files. As mentioned in Section 2, our implementation uses a CHC solver to both find relational loop invariants and discharge the resulting safety verification problems. For this purpose, our implementation leverages an enhanced version of the Spacer CHC solver [Komuravelli et al. 2016] distributed with Z3 [de Moura and Bjørner 2008]. [7]

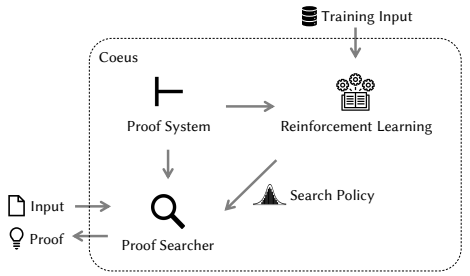


Fig. 7. Coeus architecture

8 EVALUATION

We evaluate the proposed approach by designing a series of experiments that address the following questions:

(1) How does our proposed approach perform compared to state-of-the-art relational verification tools?
(2) What is the impact of using the learned policy during search?
(3) What is the impact of backtracking search compared to directly sampling proof strategies from the learned policy?
(4) What kinds of policies does Coeus learn?
(5) What is the impact of training on the success of the learned policy?

To answer these questions, we evaluate Coeus on two different benchmark suites and compare it against several baselines. For all experiments, we set a time limit of 300 seconds and a memory limit of 10GB for the proof search algorithm, and we set a time limit of 15 seconds per each CHC solver invocation. All experiments are conducted on an Arch Linux workstation with an Intel Xeon E5-2630 CPU (2.6GHz) and 64GB of RAM.

8.1 Translation Validation Benchmarks

In our first experiment, we evaluate our approach in the context of *translation validation* [Pnueli et al. 1998]. Specifically, we use Coeus to check the correctness of various transformations performed by the ROSE compiler infrastructure [Quinlan and Liao 2011] from the Lawrence Livermore Laboratory.

[7] Similar to the SeaHorn verifier [Gurfinkel et al. 2015], our implementation augments Spacer by incorporating a Houdini-style algorithm [Flanagan and Leino 2001].

For this experiment, we consider five (intra-procedural) transformation passes from ROSE. These transformations include loop unrolling, loop splitting, loop fission, constant propagation, and partial redundancy elimination. Given an original C program P, we obtain multiple transformed programs by applying all possible combinations of these transformations to P and then use Coeus to check equivalence between P and its transformed versions.

Training set. Recall that Coeus has an offline training phase that is used for learning an optimal search policy via reinforcement learning. Towards this goal, we wrote a simple program generator that produces random, self-contained C functions. For each randomly generated program P, we obtain multiple transformed programs $P_1, \ldots, P_n$ as described above and use each (P, P_i) pair as a training example. Using this methodology, we trained Coeus on a total of 400 translation validation benchmarks. To give the reader some idea about the impact of training, Figure 8 plots the success rate of the learned policy against the number of training iterations. As we can see from this figure, the policy gradually adapts itself to better solve the problems in our training set.

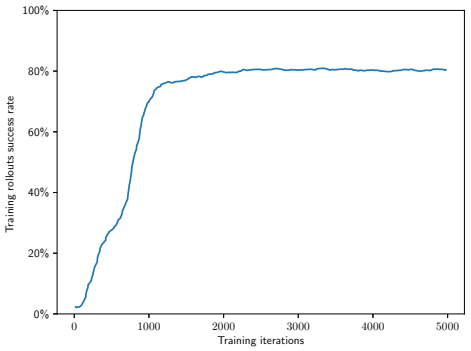


Fig. 8. Training performance on translation validation training benchmarks

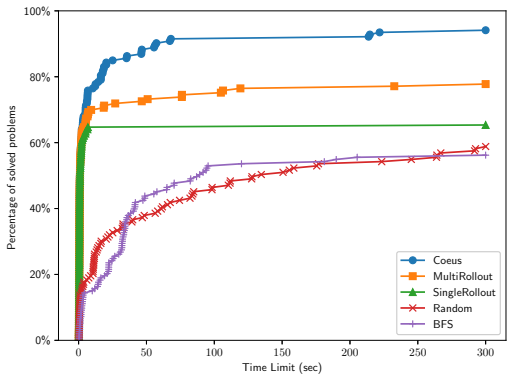


Fig. 9. Results for translation validation

Test set. The programs in our test set come from 80 functions collected from popular open-source C programs (e.g., OpenSSL, curl, etc.) that are available on Github. Given an original function `proc` from one of these applications, we apply a combination of ROSE transformations to obtain a new program `proc'`. After eliminating duplicates, we obtain a total of 153 translation validation benchmarks (i.e., pairs of programs) for our test set.

Results. Figure 9 summarizes the results of our evaluation on the translation validation domain. The x-axis shows the time limit per benchmark, and the y-axis shows the percentage of benchmarks that can be solved within that time limit. Different graphs in the figure correspond to the following variants of Coeus:

- The blue line (with circles) is the full Coeus system.
- The orange and green lines (with squares and triangles respectively) correspond to variants of Coeus that use the learned policy but not our proposed search algorithm. Specifically, Single-Rollout only explores a single rollout of the learned policy and Multi-Rollout samples multiple rollouts until a time limit is reached.
- Both the red graph (with crosses) and the purple graph (with pluses) correspond to variants that do *not* use the learned policy to guide search. The first variant (labeled Random) uses our search algorithm with a randomly generated policy, and the latter variant (BFS) uses breadth-first search.

Table 1. Comparison with other relational verification tools on translation validation benchmarks .

	Tools		
	Coeus	Descartes	VeriMap
Number of benchmarks	153		
Number of benchmarks supported by each tool	153	153	23
Number of solved benchmarks	144	77	17
Solved benchmarks / All benchmarks	94.1%	50.3%	11.1%
Solved benchmarks / Supported benchmarks	94.1%	50.3%	73.9%
Number of commonly supported benchmarks	23		
Number of solved commonly supported benchmarks	23	20	17
Solved commonly supported benchmarks / Commonly supported benchmarks	100%	87%	73.9%
Average running time for solved benchmarks (sec)	10.9	12.3	32.29

One of the key conclusions to draw from Figure 9 is that policy-guided search significantly boosts the percentage of benchmarks that can be solved within a given time limit. In particular, both BFS and Random solve less than 58% of the benchmarks within a 5 minute time-limit whereas Coeus can solve 88.9% of the benchmarks within the same limit. The second important conclusion is that our proposed search algorithm allows us to effectively utilize the learned policy. Specifically, the Single-Rollout and Multi-Rollout variants plateau at 67% and 73% respectively, whereas Coeus can continue to solve more benchmarks as we increase the time limit.

Comparison against other tools. In addition to comparing Coeus against its own variants, we also compare it against two state-of-the-art relational verification tools, namely VeriMap [De Angelis et al. 2016b] and a re-implementation of Descartes [Sousa and Dillig 2016]. VeriMap is a relational verification tool that uses a method called *predicate pairing* for solving constrained Horn clauses that arise in relational proofs. In contrast, Descartes is based on the CHL program logic and performs heuristic-guided backtracking search over the CHL proof rules. Since the original version of Descartes is for Java programs, we re-implemented a version of Descartes for C that uses the same proof rules and search heuristics.

As summarized in Table 1, Coeus outperforms both VeriMap and Descartes. Specifically, VeriMap can solve only 11% of these benchmarks within the 5 minute time limit. Upon further inspection, the low success rate of VeriMap is in part because the benchmarks constains features (e.g., bitvectors, multi-dimensional arrays) that are not supported by this tool. Nevertheless, even if we exclude 130 out of 153 benchmarks that are not supported by VeriMap, Coeus still performs significantly better: VeriMap solves 17 out of the these 23 benchmarks, whereas Coeus solves 22 out of 23. Finally, Coeus also substantially outperforms Descartes: the success rate of Descartes on the full benchmark set is around 50.3%, compared to 88.9% for Coeus.

Bugs found in ROSE. During the process of running this experiment, Coeus uncovered two sources of unsoundness in the ROSE compiler. Specifically, since the accuracy of Coeus on the training set was initially quite low, we manually inspected the benchmarks that could not be verified using Coeus. Our inspection revealed two subtle bugs in the loop unrolling and fission transformation passes implemented in ROSE. Note that the results shown in Figure 9 are obtained after fixing the loop unrolling bug and filtering out benchmarks that trigger the source of unsoundness in the loop fission pass. [8]

[8]We did not fix the latter bug since it did not seem to admit an easy fix.

8.2 Multiple Programs Written by Humans

In our previous evaluation, we considered pairs of programs where one of the programs is obtained by automatically transforming the other. In this section, we consider a slightly more challenging scenario for relational verification in which both programs are written by humans. Specifically, for this experiment, we collected pairs of manually-written programs by considering different solutions to programming challenge problems from LeetCode and HackerRank as well as pairs of programs considered in previous work [De Angelis et al. 2016b]. Furthermore, these benchmarks involve multiple different relational properties, including equivalence, non-equivalence, conditional disequality (i.e., if inputs satisfy some relationship, then outputs should be different) etc. In total, we consider 292 relational verification benchmarks and split them into training and test sets as follows: Programs with size smaller than a certain threshold are used for training, whereas the larger programs are used for testing.

This approach gives us a training set consisting of 186 benchmarks, and a test set consisting of 106. By splitting the benchmark in this way, we demonstrate how our learning-based search algorithm can generalize from the smaller examples seen during training to more complex and challenging benchmarks in the test set.

As we can see from Figure 10, the training phase shows a similar trend as in the first experiment. In particular, the accuracy is initially quite low and steadily improves until approximately 1000 training iterations, after which it seems to plateau at around 65%.

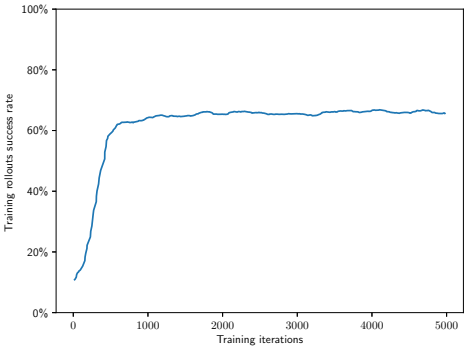


Fig. 10. Training performance for second experiment

Results. Figure 11 compares the performance of Coeus on the *testing set* with several baselines. As in the previous subsection, the x-axis shows the time limit per benchmark, and the y-axis shows the percentage of benchmarks that can be solved within that time limit. Also as before, the different graphs from Figure 11 correspond to the Multi-Rollout, Single-Rollout, Random, and BFS variants of Coeus.

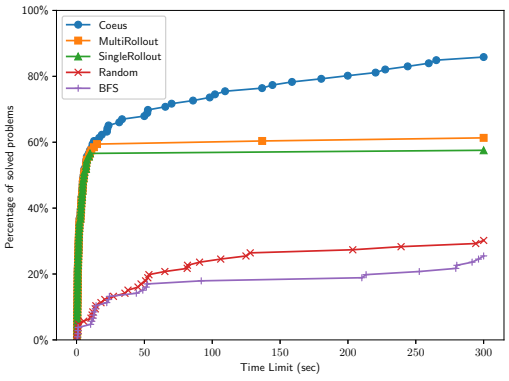


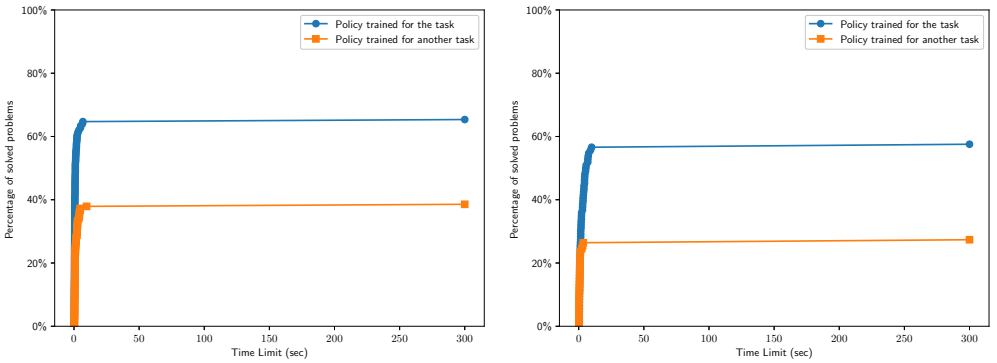
Fig. 11. Comparison on test set

The trend we see in Figure 11 largely follows the one in Figure 9. Specifically, we observe that Coeus performs significantly better than both BFS and Random, highlighting the importance of guiding search using the RL policy. We also observe that Coeus can solve significantly more benchmarks compared to Single-Rollout and Multi-Rollout as we increase the time limit, and this pattern is even more pronounced on this dataset compared to the translation validation benchmarks. This observation corroborates our hypothesis that our policy-guided search algorithm from Section 6 allows us to use the policy much more effectively.

Comparison against other tools. As in Section 8.1, we also compare the performance of Coeus against Descartes and VeriMap on this benchmark set. As shown in Table 2, Coeus solves

Table 2. Comparison with other relational verification tools on second set of benchmarks.

	Tools		
	Coeus	Descartes	VeriMap
Number of benchmarks	106		
Number of benchmarks supported by each tool	106	79	65
Number of solved benchmarks	91	44	35
Solved benchmarks / All benchmarks	85.8%	41.5%	33.0%
Solved benchmarks / Supported benchmarks	85.8%	55.7%	53.8%
Number of commonly supported benchmarks	52		
Number of solved commonly supported benchmarks	48	37	23
Solved commonly supported benchmarks / Commonly supported benchmarks	92.3%	71.2%	44.2%
Average running time for solved benchmarks (sec)	33.9	16.8	66.52



(a) Benchmarks used in section 8.1 (b) Benchmarks used in section 8.2

Fig. 12. Running Coeus with policies learned from different tasks

significantly more benchmarks compared to the other tools. Specifically, VeriMap and Descartes solve 35 and 44 of the 106 benchmarks respectively, whereas Coeus solves 93. Furthermore, if we exclude benchmarks that contain features not supported by either Descartes or VeriMap, we find that Coeus solves 92.3% of the benchmarks, whereas VeriMap solves 44.2% and Descartes solves 71.2%. We believe these results demonstrate that our proposed approach improves the state-of-the-art in relational verification.

8.3 Discussion

In this section, we discuss a number of additional aspects of our algorithm, including (i) evaluating the advantages of being data-driven, (ii) analyzing the learned properties, and (iii) explaining why using policy-guided search may outperform the single-rollout policy.

Advantages of being data-driven. An important advantage of our proposed approach is that it is *data-driven*. In particular, a key challenge for traditional verification tools is that different problem domains typically require different sets of search heuristics. Thus, to improve performance, a user must manually design search heuristics tailored to their specific domain of interest. This process can be challenging since it requires that the user is an expert both in their application domain and also in the internal workings of the verification algorithm (e.g., the underlying CHC solver). In contrast, given training data that is representative of a target domain, Coeus automatically learns a policy that works well specifically for that domain.

We empirically evaluate the advantages of being data-driven by applying the policy from Section 8.1 to the benchmarks from Section 8.2 and vice versa. In particular, in Figure 12a, we show two single-rollout performance curves of Coeus on the translation validation benchmarks. Here, the blue dotted line represents the performance of Coeus if it uses a policy trained for the translation validation task, whereas the orange line (with squares) represents the performance of Coeus if it uses a policy trained on the hand-written programs from Section 8.2. As we can see from the large gap between the blue and orange lines, Coeus performs significantly better when using a policy that has been trained on translation validation benchmarks. We also see this trend in Figure 12b: the policy trained on the translation validation benchmarks performs significantly worse when evaluated on the benchmarks from Section 8.2. Overall, we believe these results demonstrate the usefulness of learning *data-driven* relational proof search strategies that are able to *automatically* infer domain-specific insights and leverage them to boost performance.

Analysis of learned policies. We examine the policies learned by Coeus to better understand the domain-specific insights that they have inferred. Recall from Section 5 that Coeus represents policies using neural networks, which are notoriously difficult to interpret [Towell and Shavlik 1992]. To better understand the policies learned by Coeus, we approximated each of the two neural networks (i.e., representing the policies from Sections 8.1 & 8.2) using decision trees, and then manually inspect these trees. [9] Based on this analysis, we made the following observations:

- The policy learns to prioritize rules that minimize the proof length for loop-free code.
- For our first benchmark (translation validation) in Section 8.1, the policy learns to ignore proof rules that are not relevant to the kinds of transformations that ROSE performs.
- For our first benchmark, the policy learns to unroll loops when unrolling would equalize the number of loop iterations. For our second benchmark, unrolling is picked less often since it is turns out not to be very useful for the training examples in Section 8.2.
- For our second benchmark in Section 8.2, when encountering a loop in one program and function call in the other, the policy often converts the loop into a tail-recursive procedure.

Despite these fairly intuitive patterns that we have uncovered from the decision tree, we also found that the learned policy is actually quite complex. In particular, it takes a decision tree of depth more than 7 to reasonably imitate the intricate behavior of the neural network policy. Furthermore, it is worth noting that the learned policy may perform actions that are quite unintuitive and that seem to be correlated with the quirks of the underlying CHC solver. For instance, we find cases where the underlying CHC solver can much more easily discharge the resulting VCs if two independent statements are swapped in certain kinds of situations. Surprisingly, the reinforcement learning algorithm picks up on such quirks of the underlying safety checker. Thus, Coeus is able to infer unintuitive heuristics that a human would be unlikely to devise.

Impact of search. As described previously, Figure 11 shows that the policy-guided proof search algorithm substantially outperforms using the policy alone. We give an example that demonstrates the benefits of our search algorithm. In particular, Figure 13 shows one of the equivalence checking examples from Section 8.2. A successful proof strategy for this problem is to unroll the while loop in `tree0()` and then synchronize it with the for loop in `tree1()`, since equalizing the iteration counts of the two loops will drastically reduce the difficulty of solving the generated verification conditions. However, our learned policy does not favor this strategy—instead, it first tries to "synchronize" the loops directly and fails to prove equivalence, since the underlying CHC solver is not able to

[9] Note that this analysis does not consider the effects of using the learned policy in the context of a backtracking search algorithm.

```
int tree0(int n) {
  assume(n >= 0 && n <= 60);
  int h = 1; int turn = 0;
  while (n > 0) {
    if (turn == 0) {
      h = h * 2; turn = 1;
    } else {
      h++; turn = 0;
    }
    n--;
  }
  return h;
}
```

```
int tree1(int n) {
  assume(n >= 0 && n <= 60);
  int i, x = 1;
  if (n != 0) {
    for (i = 1; i <= n; i++) {
      x = x * 2; i++;
      if (i % 2 == 0 && i <= n)
        x = x + 1;
    }
  }
  return x;
}
```

Fig. 13. Example benchmark programs which require loop unrolling to verify

discharge the VCs. Nevertheless, our search algorithm progressively explores other proof strategies and discovers the right strategy after 12 failed proof attempts.

9 RELATED WORK

In this section, we survey the related work on relational verification, reinforcement learning, and the use of machine learning in programming languages.

Relational verification. As stated in Section 1, relational verification problems are typically solved by reducing them to standard safety in one of several ways. Some approaches construct a new program that is safe iff the original relational verification problem is valid [Barthe et al. 2011, 2016, 2004; Eilers et al. 2018]. Other approaches [Barthe et al. 2012; Benton 2004; Chen et al. 2017; Sousa and Dillig 2016] propose program logics for decomposing the relational verification task into a set of Hoare triples. Finally, some techniques [De Angelis et al. 2016b; Felsing et al. 2014; Mordvinov and Fedyukovich 2017] directly encode the relational verification problem as a set of constrained Horn clauses and propose new CHC solving techniques to deal with the resulting constraints [De Angelis et al. 2016b; Mordvinov and Fedyukovich 2017]. While these approaches define the space of strategies for reducing relational verification to safety checking, they do not propose algorithms for efficiently exploring the large search space. In contrast, the main contribution of this paper is to use reinforcement learning to guide proof search.

k-safety. Several papers [Chen et al. 2017; Clarkson and Schneider 2010; Sousa and Dillig 2016; Terauchi and Aiken 2005] address k-safety verification, where the goal is to prove the absence of an unintended interaction between k runs of the same program. Generally speaking, k-safety properties can be viewed as a special kind of relational verification problem, where the programs under analysis are all identical. While the ideas proposed in this paper are, in principle, applicable to proving k-safety for arbitrary values of k, our current prototype only handles 2-safety properties.

Machine learning for PL. There have been several recent successes in applying (supervised) machine learning to programming languages research. For example, machine learning has been used to infer program invariants [Padhi et al. 2016; Sharma and Aiken 2014; Sharma et al. 2013], improve program analysis [Liang et al. 2011; Mangal et al. 2015; Raghothaman et al. 2018; Raychev et al. 2015] and synthesis [Balog et al. 2016; Feng et al. 2018, 2017; Kalyan et al. 2018; Lee et al. 2018; Raychev et al. 2016b; Schkufza et al. 2013, 2014], build probabilistic models of code [Bielik et al. 2016; Raychev et al. 2016a, 2014], infer specifications [Bastani et al. 2017, 2018b; Beckman and Nori 2011; Bielik et al. 2017; Heule et al. 2016; Kremenek et al. 2006; Livshits et al. 2009], test software [Clapp et al. 2016; Godefroid et al. 2017; Liblit et al. 2005], and select lemmas for automated

theorem proving [Irving et al. 2016; Wang et al. 2017]. However, these approaches treat the selection of promising lemmas as a one-shot problem rather than a sequential decision making problem.

Reinforcement learning for PL. There has been recent interest in applying reinforcement learning (RL) to solve challenging PL problems where large amounts of labeled training data are either not available or too expensive to obtain. For instance, Si et al. use policy gradient to infer loop invariants [Si et al. 2018a]; Singh et al. apply RL improve polyhedral analysis by choosing parameters to approximate the join transformer[Singh et al. 2018], and Si et al. use reinforcement learning for program synthesis [Si et al. 2018b]. Among these techniques, the first and third one both focus on a specific problem and do not attempt to learn across different problem instances, as we do in our setting. In contrast to Singh et al, the reinforcement learning problem in our setting is more challenging, as we do not observe rewards until the very end of a rollout.

RL for game playing. Our work bears some similarities to the use of RL in game playing [Guo et al. 2014; Silver et al. 2016, 2017]. These techniques simulate different games (rollouts) under the assumption that the opponent follows the same (probabilistic) strategy and then evaluate each move based on the outcome of these simulations. In contrast to our method, these approaches all use Q-learning, which, as discussed in Section 5.4, requires mapping distinct states to distinct feature vectors. While there are mature techniques for doing this in the context of game playing, it is unclear how to featurize relational proof strategies in a way that avoids the perceptual aliasing problem.

10 CONCLUSION

We have proposed a new relational verification algorithm that uses a policy learned using reinforcement learning to guide relational proof search. We have shown how to formulate the relational verification problem as a Markov decision process and proposed a variant of the policy gradient algorithm to find an optimal policy for this MDP. Finally, we have shown how to use the learned policy to guide proof search. Experiments performed using our prototype, Coeus, show that Coeus outperforms state-of-the-art relational verification tools and demonstrate the usefulness of policy-guided proof search: Overall, Coeus solves 229 out of 259 relational verification problems in our benchmark suite, while Descartes and VeriMap solve just 121 and 52, respectively. Our experiments also highlight the importance of combining learning and backtracking search.

While some of the ideas proposed in this work could potentially be applicable to other proof search problems beyond relational verification, we believe that the proposed approach is particularly well-suited for relational verification: First, there are a large number of candidate proof rules that can be applied at each state, and good search heuristics are domain-dependent and non-trivial to design. Second, despite the large size of the search space, the models used for relational verification only need to choose between n types of available in actions (i.e., which proof rule to apply). In contrast, other proof search settings may require synthesizing auxiliary lemmas or inductive invariants that are not fixed a priori. In future work, we plan to explore the use of RL in more general proof search settings (e.g., in theorem provers like Coq and Isabelle).

A APPENDIX

A.1 Proof of Theorem 5.8

First, we show that the mapping from policies π to distributions $p^{(\pi)}$ is invertible:

Lemma A.1. *Given a distribution p over complete proof strategies, we have $p^{(\pi)} = p$, where*

$$\pi(S, A) = \frac{\sum_{S' \leq^* P(S,A)} \tilde{p}(S')}{\sum_{S' \leq^* S} \tilde{p}(S')}.$$

Proof. First, because transitions are deterministic, we have

$$p^{(\pi)}(S) \propto \sum_{\zeta} \mathbb{I}[S_T = S] \cdot p(S_0 \mid \mathcal{S}_0) \cdot \prod_{i=0}^{T-1} \pi(S_i, A_i),$$

where $p(S_0 \mid \mathcal{S}_0)$ is the probability that the initial state is S_0. Furthermore, note that there is a unique way of constructing any given complete proof strategy $S \in \mathcal{S}_F$ using actions $A \in \mathcal{A}$. Letting $\zeta_S = ((S_0, A_0, R_0), ..., (S_T, \varnothing, R_T))$ denote the unique rollout with terminal state $S_T = S$, we have

$$p^{(\pi)}(S) = p(S_0 \mid \mathcal{S}_0) \cdot \prod_{i=0}^{T-1} \pi(S_i, A_i).$$

Expanding the right-hand side, we have

$$\begin{aligned} p^{(\pi)}(S) = p(S_0 \mid \mathcal{S}_0) \cdot \prod_{i=0}^{T-1} \frac{\sum_{S' \preceq^* P(S_i, A_i)} p(S')}{\sum_{S' \preceq^* S_i} p(S')} &= p(S_0 \mid \mathcal{S}_0) \cdot \prod_{i=0}^{T-1} \frac{\sum_{S' \preceq^* S_{i+1}} p(S')}{\sum_{S' \preceq^* S_i} p(S')} \\ &= p(S_0 \mid \mathcal{S}_0) \cdot \frac{\sum_{S' \preceq^* S_T} p(S')}{\sum_{S' \preceq^* S_0} p(S')}. \end{aligned}$$

Note that the numerator of the last line equals $p(S_T)$, since the only S' such that $S' \preceq^* S_T$ for a complete state S_T is S_T itself. Similarly, the denominator equals $p(S_0 \mid \mathcal{S}_0)$, since the sets of states $\{S' \mid S' \preceq^* S_0\}$ are disjoint for different initial states S_0. In other words, $p^{(\pi)}(S) = p(S)$, as claimed. □

As a consequence, the space over policies (which reinforcement learning algorithms optimize over) and the space of distributions (which (1) optimizes over) are equal. Next, we prove that given a policy π, its cumulative reward of π equals the objective (1) evaluated at $\tilde{p} = p^{(\pi)}$:

Lemma A.2. *For any policy π for $\mathcal{M}_{proof}$, we have*

$$\mathfrak{R}^{(\pi)} = Pr_{t \sim \mathcal{T}, S \sim p_t^{(\pi)}}[O(S)],$$

where $p_t^{(\pi)}$ is the distribution $p^{(\pi)}$ conditioned on task t:

$$p_t^{(\pi)} = p^{(\pi)} \mid S \text{ is labeled with the initial proof goal for } t.$$

Proof. Note that since complete proofs are terminal states, and we only obtain reward on successful proofs (which are complete by definition). Thus, we have

$$\mathfrak{R}^{(\pi)} = \mathbb{E}_{\zeta \sim \pi}\left[\sum_{i=0}^{T} R_i\right] = \mathbb{E}_{\zeta \sim \pi}[R_T] = \mathbb{E}_{\zeta \sim \pi}[O(S_T)].$$

Finally, by definition, the distribution of S_T given a randomly sampled rollout $\zeta \sim \pi$ equals the distribution $p^{(\pi)}$. So $\mathfrak{R}^{(\pi)} = \Pr_{S \sim p^{(\pi)}}[O(S)]$, as claimed. □

The proof of Theorem 5.8 follows from Lemma A.1 and Lemma A.2.

A.2 Proof of Theorem 5.11

We can rewrite the objective $J(\theta)$ of (4) as follows:

Lemma A.3. *We have*

$$J(\theta) = \frac{1}{r+1} \sum_{i=0}^{r} \mathfrak{R}^{(\pi_{\theta,i})}.$$

Proof. Note that $\xi^{(t)}_{r,\theta}(S) = \frac{1}{r+1}\sum_{i=0}^{r} p_i$ where $p_i = p_{\pi_{\theta,i}}$. Thus, we have

$$\begin{aligned}
J(\theta) = \Pr_{t\sim\mathcal{T}, S\sim\xi^{(t)}_{r,\theta}}[O(S)=1] = \mathbb{E}_{t\sim\mathcal{T}, S\sim\xi^{(t)}_{r,\theta}}[O(S)] &= \mathbb{E}_{t\sim\mathcal{T}}\left[\frac{1}{r+1}\sum_{i=0}^{r}\mathbb{E}_{S\sim p^{(t)}_{\pi_{\theta,i}}}[O(S)]\right] \\
&= \frac{1}{r+1}\sum_{i=0}^{r}\mathbb{E}_{t\sim\mathcal{T}, S\sim p^{(t)}_{\pi_{\theta,i}}}[O(S)] \\
&= \frac{1}{r+1}\sum_{i=0}^{r}\mathbb{E}_{S\sim p_{\pi_{\theta,i}}}[O(S)] \\
&= \frac{1}{r+1}\sum_{i=0}^{r}\mathbb{E}_{\zeta\sim\pi_{\theta,i}}[O(S_T)] \\
&= \frac{1}{r+1}\sum_{i=0}^{r}\mathfrak{R}^{(\pi_{\theta,i})},
\end{aligned}$$

as claimed. □

Now, let τ denote the function by which our search algorithm constructs $\pi_{\theta,i+1}$ from $\pi_{\theta,i}$, i.e.,

$$\pi_{\theta,i} = \begin{cases} \pi_\theta & \text{if } i = 0 \\ \tau(\pi_{\theta,i-1}, \theta) & \text{otherwise.} \end{cases}$$

Then, consider the derivative of τ with respect to θ:

$$\frac{\mathrm{d}\tau}{\mathrm{d}\theta}(\pi,\theta) = \frac{\partial\tau}{\partial\pi}(\pi,\theta)\frac{\mathrm{d}\pi}{\mathrm{d}\theta} + \frac{\partial\tau}{\partial\theta}(\pi,\theta),$$

where the gradient with respect to π is the gradient with respect to the probabilities $\pi(S,A)$ of taking action A in state S. We have the following important fact about τ:

Lemma A.4. *We have*

$$\frac{\partial\tau}{\partial\pi}(\pi,\theta) = 0,$$

except on a measure zero subset.

Proof. (sketch) Our search algorithm constructs $\pi_{\theta,i}$ from $\pi_{\theta,i-1}$ by first constructing the most probable rollout $\zeta_{\max}$ according to $\pi_{\theta,i-1}$, and constructing $\pi_{\theta,i}$ deterministically from $\zeta_{\max}$ and θ, i.e.,

$$\pi_{\theta,i} = \tilde{\tau}(\zeta_{\max}, \theta).$$

In other words, $\tau(\pi,\theta) = \tilde{\tau}(\zeta_{\max},\theta)$, where $\zeta_{\max}$ is the most probable rollout according to π. However, note that $\zeta_{\max}$ is from a discrete set. Therefore, for fixed θ, τ must be a piecewise constant function of π, so the claim follows. □

Intuitively, this lemma says that the way in which we construct the sequence of policies $\pi_{\theta,0}, \pi_{\theta,1}, \ldots$ is not affected by small changes to θ. An important consequence is that

$$\frac{\mathrm{d}\tau}{\mathrm{d}\theta}(\pi,\theta) = \frac{\partial\tau}{\partial\theta}(\pi,\theta).$$

Finally, Theorem 5.11 follows directly from Lemma A.3, Theorem 5.10, and Lemma A.4.

A.3 Proof Sketch of Theorem 6.1

First, we need to introduce the notion of the *length* of a proof strategy.

Definition A.5. The length of a proof strategy Υ, written as $\mathcal{L}(\Upsilon)$, is defined as follows:

- For any proof goal $\mathcal{G}$, $\mathcal{L}(\Upsilon_0(\mathcal{G})) = 0$.
- If $\Upsilon \preceq^1 \Upsilon'$, then $\mathcal{L}(\Upsilon) = 1 + \mathcal{L}(\Upsilon')$.

Intuitively, proof length keeps track of how many proof rules have been applied. **In this paper, we only consider proof strategies of finite length.**

Lemma A.6. *Given two proof strategies* Υ_1 *and* Υ_2*, if* $\Upsilon_1 \preceq \Upsilon_2$*, then* $\ell_\pi(\Upsilon_1) \geq \ell_\pi(\Upsilon_2)$.

Proof. The lemma can be proved by induction on the difference of length between Υ_1 and Υ_2. □

Lemma A.7. *If a proof strategy* Υ *is non-failing, then for all strategy* Υ' *such that* $\Upsilon \preceq \Upsilon'$, Υ' *is non-failing.*

Proof. This lemma follows directly from the definition 3.8: for $\Upsilon = (V, E, A_\mathcal{R}, A_\varphi, A_\mathcal{G})$ and $\Upsilon' = (V', E', A'_\mathcal{R}, A'_\varphi, A'_\mathcal{G})$, if $\Upsilon \preceq \Upsilon'$, then $\bigwedge_{v \in V'} A'_\varphi(v)$ contains strictly less clauses than $\bigwedge_{v \in V} A_\varphi(v)$. If the latter is satisfiable, the former must also be satisfiable as it is strictly weaker. □

We now prove Theorem 6.1 by contradiction. Let Υ_1 and Υ_2 be two complete non-failing proof strategies and $p_\pi(\Upsilon_1) > p_\pi(\Upsilon_2)$ (thus $\ell_\pi(\Upsilon_1) < \ell_\pi(\Upsilon_2)$). Suppose Υ_2 gets dequeued from W before Υ_1 on line 5 in Algorithm 2. Since `ChooseStrategy` always picks the strategy with the smallest value of ℓ_π, we know that Υ_1 must not be in W when Υ_2 gets dequeued.

We now consider the "predecessors" of Υ_1 in the search algorithm, i.e. $\mathcal{P} = \{\Upsilon^* | \Upsilon_1 \preceq \Upsilon^*\}$. We know that Υ_1 is non-failing, so according to Lemma A.7 strategies in $\mathcal{P}$ will also be non-failing and thus will not be blocked by B on line 11 to 12. Since all proof strategies explored in Algorithm 2 refines the initial strategy $\Upsilon_0(\mathcal{G})$ for the initial goal $\mathcal{G}$, and the initial strategy is enqueued into W on line 2, there must exist one $\Upsilon^* \in \mathcal{P}$ such that Υ^* is in W when Υ_2 is dequeued.

According to lemma A.6, we have $\ell_\pi(\Upsilon^*) \leq \ell_\pi(\Upsilon_1)$. Hence $\ell_\pi(\Upsilon^*) < \ell_\pi(\Upsilon_2)$, which means that when Υ^* and Υ_2 are both in W, Υ^* will be dequeued first. This contradicts our earlier assumption that Υ_2 is dequeued before Υ^*.

A.4 Proof Sketch of Theorem 6.3

We only need to prove the proposition that when the function $\texttt{RelVerif}(\mathcal{G}, \pi, \Delta)$ returns $\perp$, every non-failing strategy must have been checked for successfulness on by Algorithm 2 on line 13. Theorem 6.3 is a direct corollary of this proposition.

The proof can be carried out by induction on the length of the non-failing strategies.

- When $\mathcal{L}(\Upsilon) = 0$, $\Upsilon = \Upsilon_0(\mathcal{G})$. The conclusion hold trivially as the initial strategy is guaranteed to reach line 13 in the first iteration of the for loop.
- Assume the proposition holds for non-failing strategies with length $n - 1$ where $n \geq 1$.
  Let $\Upsilon = \texttt{ApplyProofRule}(\Upsilon', \mathcal{R})$ with $\mathcal{L}(\Upsilon) = n$. By inductive hypothesis we know that line 13 must have been reached with $\Upsilon_i = \Upsilon'$ before. As Υ' is both not failing and not complete by definition, line 17 will be reached and Υ' will be enqueued in W.
  Now consider the iteration when Υ' gets dequeued at line 5. Line 10 is guaranteed be reached with $\Upsilon_i = \Upsilon$. Since Υ is also non-failing, it will not be blocked by B on line 11 to 12. Therefore, Υ will be checked for successfulnesss on line 13 as well.

ACKNOWLEDGMENTS

This material is based on research sponsored by DARPA award FA8750-15-2-0096 as well as NSF Award CCF-1712067. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. Any opinions, findings, andconclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. In *ICLR*.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational verification using product programs. In *International Symposium on Formal Methods*. Springer, 200–214.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2016. Product programs and relational program logics. *Journal of Logical and Algebraic Methods in Programming* 85, 5 (2016), 847–859.

Gilles Barthe, Pedro R D'Argenio, and Tamara Rezk. 2004. Secure information flow by self-composition. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*. IEEE, 100–114.

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic relational reasoning for differential privacy. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 97–110.

Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. 2018a. Verifiable reinforcement learning via policy extraction. In *NIPS*.

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *PLDI*, Vol. 52. ACM, 95–110.

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2018b. Active learning of points-to specifications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 678–692.

Nels E Beckman and Aditya V Nori. 2011. Probabilistic, modular and scalable inference of typestate specifications. In *PLDI*, Vol. 46. ACM, 211–221.

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 14–25.

Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: probabilistic model for code. In *International Conference on Machine Learning*. 2933–2942.

Pavol Bielik, Veselin Raychev, and Martin Vechev. 2017. Learning a static analyzer from data. In *International Conference on Computer Aided Verification*. Springer, 233–253.

Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. Horn clause solvers for program verification. In *Fields of Logic and Computation II*. Springer, 24–51.

Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 875–890.

Lonnie Chrisman. 1992. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *AAAI*, Vol. 1992. Citeseer, 183–188.

Lazaro Clapp, Osbert Bastani, Saswat Anand, and Alex Aiken. 2016. Minimizing GUI event traces. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 422–434.

Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (Sept. 2010), 1157–1210.

Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. 2016a. *Horn Clause Transformation for Program Verification*. Technical Report.

Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. 2016b. Relational verification through Horn clause transformation. In *International Static Analysis Symposium*. Springer, 147–169.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

Marco Eilers, Peter Müller, and Samuel Hitz. 2018. Modular Product Programs. In *European Symposium on Programming*. Springer, 502–529.

Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating Regression Verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. 349–360.

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 420–435.

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *PLDI*, Vol. 52. ACM, 422–436.

Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME '01)*. Springer-Verlag, Berlin, Heidelberg, 500–517.

Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 50–59.

Joseph A Goguen and José Meseguer. 1982. Security policies and security models. In *Security and Privacy, 1982 IEEE Symposium on*. IEEE, 11–11.

Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. 2014. Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning. In *Advances in neural information processing systems*. 3338–3346.

Arie Gurfinkel, Temesghen Kahsai, and Jorge A. Navas. 2015. SeaHorn: A Framework for Verifying C Programs (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, Christel Baier and Cesare Tinelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 447–450.

Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified synthesis: automatically learning the x86-64 instruction set. In *PLDI*, Vol. 51. ACM, 237–250.

Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. 2016. Deepmath-deep sequence models for premise selection. In *Advances in Neural Information Processing Systems*. 2235–2243.

Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. In *ICLR*.

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based Model Checking for Recursive Programs. *Formal Methods in System Design* 48, 3 (June 2016), 175–205.

Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. 2006. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 161–176.

Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Verification*. Springer, 712–717.

Shuvendu K Lahiri, Kenneth L McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential assertion checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 345–355.

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 436–449.

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115.

Percy Liang, Omer Tripp, and Mayur Naik. 2011. Learning minimal abstractions. In *POPL*, Vol. 46. ACM, 31–42.

Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. 40, 6 (2005), 15–26.

Benjamin Livshits, Aditya V Nori, Sriram K Rajamani, and Anindya Banerjee. 2009. Merlin: specification inference for explicit information flow problems, Vol. 44. ACM, 75–86.

Ravi Mangal, Xin Zhang, Aditya V Nori, and Mayur Naik. 2015. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 462–473.

R Andrew McCallum. 1993. Overcoming incomplete perception with utile distinction memory. In *Proceedings of the Tenth International Conference on Machine Learning*. 190–196.

William H Montgomery and Sergey Levine. 2016. Guided policy search via approximate mirror descent. In *Advances in Neural Information Processing Systems*. 4008–4016.

Dmitry Mordvinov and Grigory Fedyukovich. 2017. Synchronizing constrained Horn clauses. *LPAR, EPiC Series in Computing. EasyChair* (2017).

Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. In *PLDI*, Vol. 51. ACM, 42–56.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '98)*. Springer-Verlag, Berlin, Heidelberg, 151–166.

Dan Quinlan and Chunhua Liao. 2011. The ROSE Source-to-Source Compiler Infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*.

Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided program reasoning using Bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 722–735.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016a. Probabilistic model for code with decision trees. In *OOPSLA*, Vol. 51. ACM, 731–747.

Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016b. Learning programs from noisy data. In *POPL*, Vol. 51. ACM, 761–774.

Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from big code. In *POPL*, Vol. 50. ACM, 111–124.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *PLDI*, Vol. 49. ACM, 419–428.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *ASPLOS*, Vol. 41. ACM, 305–316.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. In *PLDI*, Vol. 49. ACM, 53–64.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust region policy optimization. In *International Conference on Machine Learning*. 1889–1897.

Rahul Sharma and Alex Aiken. 2014. From invariant checking to invariant inference using randomized search. In *CAV*.

Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V Nori. 2013. Verification as learning geometric concepts. In *International Static Analysis Symposium*. Springer, 388–411.

Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018a. Learning loop invariants for program verification. In *Advances in Neural Information Processing Systems*. 7762–7773.

Xujie Si, Yuan Yang, Hanjun Dai, Mayur Naik, and Le Song. 2018b. Learning a Meta-Solver for Syntax-Guided Program Synthesis. In *ICLR*.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529, 7587 (Jan. 2016), 484–489.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *Nature* 550, 7676 (2017), 354.

Gagandeep Singh, Markus Püschel, and Martin Vechev. 2018. Fast Numerical Program Analysis with Reinforcement Learning. In *International Conference on Computer Aided Verification*. Springer, 211–229.

Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Proc. Conference on Programming Language Design and Implementation*. 57–69.

Marcelo Sousa, Isil Dillig, and Shuvendu Lahiri. 2018. Verifying Semantic Conflict-Freedom in Three-Way Program Merges. *arXiv preprint arXiv:1802.06551* (2018).

Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*. 1057–1063.

Tachio Terauchi and Alex Aiken. 2005. Secure Information Flow As a Safety Problem. In *Proceedings of the 12th International Conference on Static Analysis (SAS'05)*. 352–367.

Geoffrey Towell and Jude W. Shavlik. 1992. Interpretation of Artificial Neural Networks: Mapping Knowledge-Based Neural Networks into Rules. In *Advances in Neural Information Processing Systems 4*, J. E. Moody, S. J. Hanson, and R. P. Lippmann (Eds.). Morgan-Kaufmann, 977–984.

Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. 2017. Premise selection for theorem proving by deep graph embedding. In *Advances in Neural Information Processing Systems*. 2786–2796.

Wenhan Xiong, Thien Hoang, and William Yang Wang. 2017. Deeppath: A reinforcement learning method for knowledge graph reasoning. In *EMNLP*.

Hongseok Yang. 2007. Relational separation logic. *Theoretical Computer Science* 375, 1-3 (2007), 308–334.

Anna Zaks and Amir Pnueli. 2008. Covac: Compiler validation by program analysis of the cross-product. In *FM 2008: Formal Methods*. Springer, 35–51.