Learning to Synthesize Relational Invariants

Jingbo Wang
University of Southern California
Los Angeles CA 90089, USA
jingbow@usc.edu

Chao Wang
University of Southern California
Los Angeles CA 90089, USA
wang626@usc.edu

ABSTRACT
We propose a method for synthesizing invariants that can help verify relational properties over two programs or two different executions of a program. Applications of such invariants include verifying functional equivalence, non-interference security, and continuity properties. Our method generates invariant candidates using syntax guided synthesis (SyGuS) and then filters them using an SMT-solver based verifier, to ensure they are both inductive invariants and sufficient for verifying the property at hand. To improve performance, we propose two learning based techniques: a logical reasoning (LR) technique to determine which part of the search space can be pruned away, and a reinforcement learning (RL) technique to determine which part of the search space to prioritize. Our experiments on a diverse set of relational verification benchmarks show that our learning based techniques can drastically reduce the search space and, as a result, they allow our method to generate invariants of a higher quality in much shorter time than state-of-the-art invariant synthesis techniques.

ACM Reference Format:

1 INTRODUCTION
Invariant generation is a fundamental problem in program analysis and verification, e.g., to prove that assertions always hold during program execution. Loop invariants [25, 33], for example, are conditions that must be true at the beginning and the end of every iteration of a loop. Since the problem is undecidable in general, all practical techniques must search for invariants heuristically in a potentially-infinite space of candidates. While there is a large body of work on making the search efficient, e.g., using guided search [32, 59], data-driven sampling [48, 72], supervised learning [63, 64], continuous logic network [57, 71], and decision tree with templates [27, 28], they target a single program, as opposed to relational invariants which is the main focus of this paper.

Relational invariants are logical assertions defined over multiple programs or program executions. They are useful for reasoning about the relationship between these programs or program executions, one example application is to prove functional equivalence, i.e., two programs always behave the same when given the same input [16, 61]. Another example application is to check a security property called non-interference, i.e., executing a program using two different values of a secret input does not lead to observable differences in the public output [6, 11, 13]. The third example application is to verify the continuity property, i.e., a program remains robust with respect to infinitesimal changes to the input [12]. However, to the best of our knowledge, there is still a lack of techniques and tools for efficiently synthesizing relational invariants.

State-of-the-art invariant synthesis tools, which were designed primarily for a single program, cannot be easily adapted to generate relational invariants. To confirm this, we have experimented with two state-of-the-art tools: Code2Inv [64] and LinearArbitrary [72]. In this experiment, we took two structurally-different but functionally-equivalent programs, P1 and P2, and created a merged program P that executes instructions from P1 and P2 in lockstep; then we specified the equivalence relation as \( \langle \Phi \rangle P \langle \Psi \rangle \), which is a Hoare triple [33] saying that, if \( P_1 \) and \( P_2 \) start from the same state (\( \Phi \)), after the lockstep execution, they must end at the same state (\( \Psi \)). Unfortunately, neither tools can generate invariants that are strong enough to help verify the equivalence relation. Code2Inv [64] generated an invariant in which none of the predicates was relational, while LinearArbitrary [72] generated an over-fitted solution that unnecessarily depends on some arbitrary constants appeared in the sampled data. More details of this experiment can be found in Section 2.

To overcome the limitations, we have developed a new method named Code2RelInv, whose input consists of a merged program \( P \) and a specification in the form of a Hoare triple \( \Phi = (\Phi, \Psi) \), where \( \Phi \) is the precondition and \( \Psi \) is the postcondition. The output of Code2RelInv, which is a relational invariant \( I \), is guaranteed to be both inductive (i.e., a true invariant) and sufficient (i.e., strong enough to prove the property at hand).

Figure 1 shows the overall flow of Code2RelInv, which uses a standard syntax guided synthesis (SyGuS) [2] component to generate invariant candidates \( I \), one at a time, from a hypothesis space defined by a domain-specific language (DSL). Then, it uses an SMT-solver based program verifier to check if \( I \) is both inductive and sufficient. Candidates that are not inductive, or not sufficient, are removed. The iterative process continues until a desired invariant is found, or a predetermined time limit is reached. The novel part of our method is the component that leverages the learning based techniques to reduce the search space.

We propose two learning based techniques to make the synthesis procedure efficient. The first one is logical reasoning (LR) based search space pruning: as soon as the verifier declares an invariant candidate \( I \) as invalid, we analyze the reason why it is invalid and, based on the reason, skip all other invariant candidates that share the same reason. In this sense, our method has the ability to learn...
We experimentally compared our method with a state-of-the-art invariant synthesizer, Code2Inv [4], and three types of relational properties: equivalence, inductive, and relational verification benchmarks, consisting of a set of C programs and three types of relational properties: equivalence over various loop optimizations [7], non-interference for DARPA STAC programs [4] and continuity of a number of sorting algorithms [12]. We experimentally compared our method with a state-of-the-art invariant synthesizer, Code2Inv [64]. The experimental results show that, for all benchmarks, our method was able to generate the desired invariants quickly, whereas Code2Inv failed in most cases. Furthermore, both of our learning based techniques (LR and RL) are effective in reducing the search space: with these techniques, the number of invariant candidates explored by our method can be reduced by as much as 96%.

To summarize, this paper makes the following contributions:

- We propose a new method for synthesizing relational invariants, which uses both syntax-guided synthesis (SyGuS) and an SMT solver based program verifier to guarantee that the invariants are both inductive and sufficient.
- We propose a logical reasoning (LR) based technique, which leverages the SMT solver’s ability to compute unsatisfiability cores to prune the search space.
- We propose a reinforcement learning (RL) based technique, which leverages the verifier’s results as positive and negative rewards to prioritize the search.
- We conduct experimental evaluation on a diverse set of relational verification benchmarks to demonstrate the effectiveness of our method.

2 MOTIVATION

Consider the two programs in Figure 2, taken from [61], where $P_2$ is obtained from $P_1$ using a loop optimization called strength reduction [67]: if variable $k$ is incremented in each loop iteration, the expression $k \cdot c$ can be safely rewritten as $k$, given that $c$ is a constant and increments to $k$ at each iteration are scaled by $c$. Since variables in the two programs may have different values, for each variable $x$ in $P_1$, we use $\bar{x}$ to denote the same variable in $P_2$. To prove the equivalence, an invariant must be provided to show how program states in $P_1$ and $P_2$ are related to each other.

2.1 Problem Statement

In relational verification, it is a common practice to construct a merged program $P$, shown in Figure 2 (c), that executes instructions from $P_1$ and $P_2$ in lockstep. Statements from $P_1$ and $P_2$ are carefully aligned, e.g., by adding auxiliary statements or even unrolling some loop iterations if needed. While techniques for loop alignment are important, they are not the focus of this work; for more information please refer to [7, 16].

The property under verification is expressed as $\varphi = \langle \Phi \rangle P \langle \Psi \rangle$, meaning that, from a state where the precondition $\Phi$ holds, executing $P$ leads to a state where the postcondition $\Psi$ holds. Since loops are the most challenging part in program verification, without loss of generality, we denote the merged program as $P = \text{while} g \ do \ S$. In this context, we want an invariant $I$ of the program $P$ with respect to the property $\varphi$ to satisfy three conditions:

(a) the precondition $\Phi$ implies $I$ at the beginning of the loop, denoted $\Phi \rightarrow I$;
(b) $I$ being true at the beginning of a loop implies $I$ being true at the end of the loop, denoted $\langle I \land g \rangle S \{I\}$, and
(c) \( I \) being true at the end of the loop implies the postcondition \( \Psi \), denoted \( I \land \neg g \rightarrow \Psi \).

Conditions (a) and (b) imply that \( I \) is inductive, and Condition (c) implies that \( I \) is sufficient for proving the property \( \varphi \).

### 2.2 Limitations of Existing Methods

Feeding the merged program \( P \) to state-of-the-art invariant synthesizers such as Code2Inv and LinearArbitrary does not produce the desired invariants.

For the example in Figure 2, Code2Inv [64] produces \(((\overline{\overline{I}} \leq (0 - 1) || \overline{P} : (\overline{\overline{P}} + \overline{\overline{K}})) \land (n := n || \overline{I} := (\overline{\overline{K}} + 0)))\) which is neither inductive nor sufficient. Furthermore, since Code2Inv relies on the standard program dependency information to decide whether two variables should be put into the same predicate, while pairs of variables from \( P_1 \) and \( P_2 \) do not have control/data dependencies at all, they never show up in the same predicate.

LinearArbitrary [72] produces \((x - \overline{x} \geq 0 \land x - \overline{x} \leq 0 \land (\neg (i \leq 1) \lor j < 2) \land (\neg (i \leq 2) \lor \neg (i \leq 3)) \land \ldots)\) which is overfitted in the sense that some of the predicates unnecessarily depend on constant values appeared in the sampled data. This is an undesired consequence of using techniques that learn from sampled data.

### 2.3 How Our Baseline Method Works

In contrast, our method is able to generate the desired relational invariant: \( I := \{x = \overline{x} \land k + s = \overline{k} \land i = \overline{i}\} \). Note that the invariant is both inductive and sufficient. Furthermore, the invariant is relational in that each predicate refers to a pair of program variables from \( P_1 \) and \( P_2 \), respectively.

Our method works as follows. First, we capture the space of invariant candidates using the domain specification language (DSL) shown in Figure 4. Then, we use the syntax guided synthesis (SyGuS) framework [2] to enumerate invariant candidates from the hypothesis space, one at a time, and using a verifier if they are both inductive and sufficient.

The first invariant candidate may be \( I := \{k = \overline{k} \land x = \overline{x}\} \), whose abstract syntax tree (AST) is shown in Figure 3 (a) as AST\(_0\). Here, the label 0 means the node is not-in-use (NULL). For \( I \) to be inductive, the formula below must hold:

\[
F_I := (\Phi \rightarrow I) \land (\{I \land g\} \subseteq \{I\})
\]

This is a classic program verification problem [25, 33], which can be solved by constructing a set of verification conditions (VCs) and then discharging these VCs using an SMT solver. In our method, we use Z3 [17] as the SMT solver.

For \( I \) to be sufficient, the formula below must hold:

\[
F_s := (I \land \neg g \rightarrow \Psi)
\]

We check this formula also using the Z3 SMT solver.

Since the first invariant candidate is not inductive, it will fail the check by \( F_I \). Therefore, our method generates a new invariant candidate. Without our learning based optimizations, however, the baseline SyGuS procedure would have produced the candidate shown on the left of Figure 3(b). This is not efficient because the new candidate would no only fail the check by \( F_I \), but also fail for the same reason as the initial candidate.

### 2.4 Our Learning-based Optimizations

With a logical reasoning (LR) based technique, our method is able to identify the reason why the first candidate fails to be inductive. As shown by the red dashed box in Figure 3(a), it is because the first candidate contains the conflict predicate \( \delta_k := (k = k) \). In other words, \( \delta_k \) contradicts to the program semantics. Thus, as long as a candidate contains \( \delta_k \), it will fail to be inductive.

Since the second candidate on the left of Figure 3(b) also contains \( \delta_k \), it would fail to be inductive for the same reason. Thus, our method avoids generating this candidate in the first place. Instead, it generates the candidate on the right of Figure 3(b).

In addition to the LR based optimization, our method also uses a reinforcement learning (RL) based optimization to prioritize the search. While invariant candidates are being analyzed, the RL agent uses the verifier’s results as positive and negative rewards to compute an exploration policy. The exploration policy defines, for each AST node shown in Figure 3, a probability distribution of its possible values, which can be used by the synthesizer to pick values so as to maximize the expected reward.

In the running example, assuming that the next AST node to fill is node 5 and the node type is an Arithmetic Expression \( c = \text{var} \), \( \text{par} \), we need to choose one of the two elements. By using the exploration policy computed by the RL agent, we can pick an element with a higher probability to generate the next candidate.

### 3 OUR METHOD

In this section, we present the baseline method, while leaving the LR and RL based optimizations to Sections 4 and 5, respectively.
Algorithm 1 Our method for synthesizing relational invariants.

Input: Merged program \( P \), Relational property \( \varphi \)

Output: Relational invariant \( I \)

1. \( I \leftarrow \emptyset \), \( d_l \leftarrow 1 \), and \( G \leftarrow \{(d_l, G)|1 \leq d_l \leq 2^H - 1\} \)
2. \( C \leftarrow 0 \), and \( \mathcal{P}_{RL} \leftarrow \emptyset \)
3. while running time < threshold do
   4. \( I, T \leftarrow GEN\_NEXT\_INV\_CANDIDATES(I, d_l, G, C, \mathcal{P}_{RL}) \)
   5. if PROVED\_INDUCTIVE(\( P, \varphi, I \)) then
   6. \( I \leftarrow I \cup T \)
   7. \( S_C, \lambda_C \leftarrow PRUNE\_BY\_LR(P, \varphi, I, d_l, C) \)
   8. \( PRIORITIZE\_BY\_RL(I, T, S_C, \lambda_C, \mathcal{P}_{RL}) \)

Figure 4: The DSL for relational invariants, where \( c \) is a set of constants, \( var \) is a set of variables, and \( A \) is a set of arrays.

3.1 Top-level Procedure

Algorithm 1 shows the top-level procedure which takes a merged program \( P \) and a property \( \varphi \) as input, and returns the invariant \( I \) as output. It first initializes the data structures: \( I, d_l, G, \mathcal{P}_{RL} \), and \( C \). Here, \( I \) is the AST of the invariant candidate, which is initialized to NULL. The decision level \( d_l \) is the index of the AST node in \( I \) that will be modified to generate the next invariant candidate. While modifying the AST, we follow the depth-first-search (DFS) order. Therefore, \( d_l \) refers to the backtracking point during DFS. \( G \) is a data structure that maps each backtracking point \( d_l \) to its unvisited grammar set \( G \); this is elaborated in Section 3.2. We ignore \( C \) and \( \mathcal{P}_{RL} \) for now since they implement the learning-based optimizations to be presented in Sections 4 and 5.

After initializing the data structures, our method uses syntax-guided synthesis (SyGuS) to generate an invariant candidate \( I \) in the hypothesis space defined by a domain specific language (DSL). If \( I \) is both inductive and sufficient, it will be returned as the output. Otherwise, subroutines PRUNE\_BY\_LR and PRIORITIZE\_BY\_RL are invoked to reduce the search space, before our method generates another invariant candidate.

In the remainder of this section, we focus on the baseline version of Algorithm 1 without the LR and RL based optimizations.

3.2 Domain-Specific Language (DSL)

Figure 4 shows the context-free grammar \( G \) of the DSL for expressing the invariants. \( G \) maps a type (i.e., the left-hand side of ":=") to a set of compatible values (i.e., the right-hand side of ":="). For instance, the feasible values for representing atom predicate are \( G[p] = \{a \oplus a, \odot a, A' \odot A\} \). The DSL is designed such that invariants in the DSL can be analyzed by any SMT solver that supports the linear integer arithmetic (LIA) and array theories.

Let \( var \) be the set of variables from programs \( P_1 \) and \( P_2 \), \( A \) be the set of constants, and \( c \) be the set of constants. Linear integer arithmetic expression, \( a \), is defined over \( var \) and \( c \), while array expression \( \bar{a} \) is defined over \( A \).

Function getValue(\( A, i \)) returns the \( i \)-th element of the array \( A \), while \( \lambda A \) denotes applying function \( F \) to array \( A \), which returns a single value. Here, function \( F \) may be sum, min or max, which are frequently used in programs that manipulate arrays.

Function getSubset(\( A, i, i_k \)) returns another array \( A[i, i_k] \), which has a subset of the elements. Similarly, getSubset1(\( A, i, c \)) returns a subset of the elements satisfying the condition \( (i \mid c) \). For instance, getSubset(\( A, i \neq 2 \)) returns a new array \( S = \{A[i] \mid i 
eq 2 \} \).

As an example, consider the expression \( (i = j + 1) \land (d[i, 1] = \bar{d}[i, 1]) \land (h[j] = a[j]) \land (a = \bar{a}) \). In our DSL, it is \( (i = j + 1) \land (\text{getSubset}(d, i, j), h[j] = \text{getValue}(h, j) \land (\text{getValue}(a, j) \land (a = \bar{a})) \).

3.3 Abstract Syntax Tree (AST)

We use a complete binary tree to represent the ASTs of invariant candidates. Let \( H \) be the height of the tree, the total number of nodes will be \( 2^H - 1 \). Figure 5 shows an example tree whose height is \( H = 3 \). Each node has a unique index \( N \in \{1, \ldots, 2^H - 1\} \). The index of the root node is 1. Given any node with index \( N \), its two child nodes have indices \( 2N \) and \( 2N + 1 \), respectively.

Each node \( N \) has a type \( \chi_N \), which may be \( var, c, A, \) or any element in the set \( \{\varphi, p, \bar{a}, i, a, a_0, \odot, \oplus, \ldots\} \), which corresponds to the set of grammar rules in Figure 4. If the type \( \chi_N \) is \( var, c, A \), the node \( N \) corresponds to a scalable variable in \( var \), a constant in \( c \), or an array in \( A \). Otherwise, the node corresponds to a set of production rules defined by the grammar in Figure 4.

For example, if \( \chi_N = p \), the set of production rules, \( G[\chi_N] \), is \( \{a \odot a, \bar{a} \odot a, A' \odot A'\} \). Assuming \( a \odot a \) is chosen, we have \( \chi_N = \odot(\chi_2N, \chi_2N + 1) \), meaning that the two child nodes have a type \( \chi_2 = \chi_2 + 1 \).

Thus, an invariant \( I \) can be represented by a set of node (\( N \)) and value (\( v \)) pairs:

\[
I := \{(N, v) \mid 1 \leq N < 2^H - 1, v \in G[N]\}
\]

(3)

In Figure 5, for example, we have an incomplete invariant under construction \( I_1 = ((1, \&\&), (2, \_), (4, var1)) \).

3.3.1 Constructing an AST.

Our baseline method systematically traverses all ASTs that can be represented by the binary tree. To simplify implementation, the traversal strictly follows the DFS order. For the example in Figure 5, the DFS order is \( L = [1, 2, 4, 5, 3, 6, 7] \). Similarly, for the example in Figure 4, the DFS order is \( L = [1, 2, 4, 8, 9, 5, 10, 11, 3, 6, 12, 13, 7, 14, 15] \).

Figure 5 illustrates the construction of an AST rooted at Node 1. Assume that all nodes have the initial value 0, meaning they are not yet part of the AST. Furthermore, assume the root node has the type \( \varphi \), meaning it is a Boolean expression. Our method starts with Node 1. If it assigns the operator "\&\&" to Node 1, the tree maps to \( I_1 := \{(1, \&\&)\} \). According to the DSL in Figure 4, the child node types must be \( \chi_2 = \chi_3 = \bar{\varphi} \).

Our method continues with Node 2. If it assigns the operator "\_" to Node 2, the tree maps to \( I_2 := \{(1, \&\&), (2, \_), (4, var1)\} \). According to the DSL, the child node types are \( \chi_4 = \chi_5 = a \). By following the DFS order, our method fills the entire tree, to obtain the invariant.
The subroutine `filtering techniques` to optimize the baseline method, to get rid of

3.4.1 Syntactic Filtering. We have implemented several syntactic filtering techniques to optimize the baseline method, to get rid of candidate. Some nodes may remain \( \emptyset \), meaning they are still not part of the AST.

3.4.2 Modifying an AST. Figure 3 illustrates the construction of the next AST by modifying the current AST. For now, let us focus on the two ASTs on the left-hand side, since they correspond to the baseline. Here, \( AST_1 \) is the current AST, and \( AST_{1+1} \) is the next AST that our method generates. According to the DFS order, if the backtracking point \( (N_{\text{dfl}}) \) is 7, we should modify Node 7.

Since Node 7 is of the type \( \kappa_2 = a_0 \), which may be either \( var \) or \( c \star var \), we change Node 7 from \( var \) to \( c \star var \). This results in assigning the operator \( \kappa \) to Node 7 and then assigning values to the child nodes accordingly. The new backtracking point is set to \( N_{\text{dfl}} = 15 \).

3.4 Generating Invariant Candidates

We now present the subroutine `GEN_NEXT_INV_CANDIDATE`, shown in Algorithm 2. Let us ignore the brown colored statements for now, since they are specific to our RL based optimization.

Given the current candidate \( I_{\text{old}} \), and the backtracking level \( dl \), the subroutine retains the values of all nodes in \( L[0 : dl - 1] \) and regenerates values of the remaining nodes as follows. First, for the node \( L[dl] \), it picks a value that has not yet been visited by this node, and then labels this value as visited in its grammar set \( G[X_{dl}] \) (Lines 7-8). Then, starting from \( L[dl] \), it creates an AST rooted at \( L[dl] \) by recursively applying the production rules in Figure 4.

At the end, it computes the new backtracking level \( dl \). If there are still unvisited values in \( G[X_{dl}] \), where \( G = G[dl] \), the backtracking level remains unchanged. Otherwise, it becomes the last \( dl \) where \( G[X_{dl}] \) contains unvisited values. After the new backtracking level is found, our method also resets the grammar set as unvisited for all levels in between.

Whenever the RL based optimization is enabled, the brown colored statements will be executed. There are two main differences between this version and the baseline. First, instead of traversing the ASTs in a strict DFS order, it picks the value \( v \) by sampling according to a probability distribution given by \( \mathcal{F}_{RL} \) (computed by the RL agent), and documents the history \( (I, v, 0) \) in a trace \( \mathcal{T} \). Second, at the end of the procedure, instead of backtracking based on the strict DFS order, it always backtracks all the way to \( dl = 1 \).

3.4.1 Syntactic Filtering. We have implemented several syntactic filtering techniques to optimize the baseline method, to get rid of

![Figure 5: Step-by-step construction of invariant candidate.](image-url)
output, the procedure also updates two global data structures \( C \) and \( dl \), where \( C \) is the accumulative set of all conflict predicates generated so far, and \( dl \) is the backtracking level.

In the remainder of this section, we present our method for constructing the UNSAT core \( S_C \), computing the conflict predicate \( \delta_c \), performing non-chronological backtracking (by changing \( dl \)), and computing the strengthening predicate \( \delta_s \).

4.1 Constructing the UNSAT Core

We take the inductive part of \( FT \) for demonstration. \( FT := \forall v. \{ I (v) \land g(v) \} S (v, v') \{ I (v') \} \). In a loop’s body \( S, o \) stands for old variables (incoming to the loop) and \( o' \) for new ones (outgoing from the loop), e.g., a statement \( x = x + 1 \) in a loop’s body is encoded as \( x' = x + 1 \) in an SMT formula.

To identify the reason why a candidate fails, we leverage the SMT solver’s capability of extracting UNSAT cores from an unsatisfiable formula. However, this is not straightforward because formulas \( FT \) and \( F_S \) which are used for verification, contain universal quantifier (\( \forall \)), and when they fail verification, the SMT solver returns satisfying solutions for the negated formulas \( \neg FT \) and \( \neg F_S \). However, to generate UNSAT cores, there must be unsatisfiable formulas to start with. Thus, the question is how to construct unsatisfying formulas from these two satisfying formulas?

Counterexamples. Consider \( \neg FT \). When it is evaluated as SAT, the solver returns a model, consisting of values assigned to the variables that makes \( I \) fail the verification. While it may be tempting to infer the root cause of the failure from this specific model, the result would be unsound in general, and most likely would not make sense in practice. This is because the model may be inconsistent with the precondition \( \Phi \) in the relational specification. In fact, checking if the model can be derived by the precondition \( \Phi \) would require the construction of a long series of recursion-free unwindings [72].

In this work, we propose a novel technique to overcome the aforementioned challenges. Our method relies on constructing a so-called mirror formula, \( \neg M_F \), such that \( \neg M_F \) being unsatisfiable implies that \( FT \) does not hold and candidate invariant \( I \) is invalid. Therefore, we can use the formula \( \neg M_F \) to extract the UNSAT core. However, it is worth noting that the reverse does not have to be true. The invalidity of \( I \) does not imply the unsatisfiability of \( \neg M_F \).

4.1.1 The Mirror Formula.

**Definition 1 (Mirror Formula).** Assuming the verification problem requires the validity of the \( FT \), defined as the Hoare triple \( \forall v. \{ I (v) \land g \} S (v, v') \{ I (v') \} \), the mirror formula \( \neg M_F \) is defined as \( \forall v. \{ I (v) \land g \} S (v, v') \{ \neg I (v') \} \).

Whenever \( FT \) fails to be verified, we check if \( \neg M_F \) is unsatisfiable. If \( \neg M_F \) is indeed unsatisfiable, we use the UNSAT core extracted from \( \neg M_F \) to identify the root cause, which in turn can guide us to prune the search space. Using the mirror formula to explain the root cause of an invalid formula \( FT \) is sound in that, as long as an explanation can be found in this way, it is guaranteed to be the root cause. This is stated in the following theorem.

**Theorem 2 (Soundness of \( \neg M_F \)).** Given an invariant candidate \( I \) and the corresponding \( FT \), the unsatisfiability of its mirror formula, \( \neg M_F \), implies the invalidity of \( I \).

---

**Algorithm 3** Our LR based search space pruning.

1. **procedure** \( \text{PRUNE\_BY\_LR}(P, \phi, I) \)
2. **if** \( \text{CheckUnsat}(\neg M_F) \) **then**
3. \( S_C := \text{ObtainUnsatCore}(P, \phi, I) \)
4. \( \delta_c := \text{UpdateTraverseOrder}(S_C, I, C) \)
5. **else**
6. \( S_C, \delta_c := 0, \emptyset \)
7. \( \delta_s := \text{ObtainAbductPred}(P, \phi, I) \)
8. **if** \( \text{CheckFeasible}(\delta_s, I) \) **then**
9. \( I := I \land \delta_s \)
10. **if** \( \text{ProvedInductive}(P, \phi, I) \) **then** \( I_i := I \)
11. **return** \( S_C, \delta_c \)
12. **procedure** \( \text{UpdateTraverseOrder}(S_C, I, T_{NOW}, C) \)
13. \( \delta_c := S_C \land \{ \delta_l | \delta_l \in I \} \)
14. \( C := C \cup \delta_c \)
15. \( M_f := \{ n_i, n_i + 1 \} \leq n_i \leq 2^H - 1, \delta_l \subseteq AST \}
16. \( n_e := \text{GetValueByKey}(M_f, \delta_c) \) \( \triangleright \) conflict predicate
17. \( dl := n_e \)
18. **return** \( \delta_c \)

We provide the following formal proof to describe the intuition behind the theorem 2, which illustrates the relationship between \( I \) and \( \neg M_F \).

Our key insight is to come up with a negated formula such that when it is UNSAT, it implies that the invariant is invalid. According to Definition 1, \( M_F := \forall v. \{ I (v) \land g \} S (v, v') \{ \neg I (v') \} \), if \( M_F \) is satisfiable, then all its conjuncts, including \( \neg I (v') \) evaluate to true. Consequently, if \( M_F \) is satisfiable then \( I (v') \) is false, i.e., the invariant \( I \) is invalid. Since \( M_F \) is universally quantified, the solver evaluates its negated form \( \neg M_F \), and when it is UNSAT, it means that non-negated one is SAT, and hence, \( I \) is invalid. Now, UNSAT cores can be extracted from \( \neg M_F \) to prune the search space.

This approach catches only some cases for \( I \) being invalid, i.e., when \( I \) does not work with the fresh variables \( I (v') \). There could be cases when \( FT \) fails on its other conjuncts, e.g., on \( I (v) \), but the mirror formula won’t be able to detect those cases. Specifically, \( I (v) \) may not be strong enough to imply \( I (v') \). We will elaborate how we handle this scenario in Section 4.3.

4.1.2 The UNSAT Core Example.

For the motivating example in Figure 2, when the inductive condition, \( FT \), fails to be verified for the first invariant candidate shown in Section 2.3, our method constructs the mirror formula and then computes the UNSAT core:

- \( a_{13}: kN = k + 1 \land a_{14}: kN = k + 5 \land a_{15}: (i < 5 \land kNN = kN) \lor (i \geq 5 \land kNN = kN + 3) \)
- \( a_{17}: (i < 5 \land kNN = kN) \lor (i \geq 5 \land kNN = kN + 15) \land a_{21}: kNN = kNN \)

For ease of presentation, the program variables are shown in the static single assignment (SSA) format: \( kN \) represents the updated version of \( k \) and \( kNN \) represents the updated version of \( kN \).

Inside this UNSAT core, only \( a_{21} \) is from the invariant candidate \( I \), while the rest of the constraints in the UNSAT core encodes the program semantics. Therefore, our method labels \( a_{21} \) as the conflict predicate \( \delta_e \), highlighted by the red dashed box on the right side of Figure 3(a). In other words, any invariant candidate that contains \( \delta_e \) is guaranteed to fail verification for the exact same reason.
4.2 Non-chronological Backtracking

We now discuss how the UpdateTraversalOrder procedure in Algorithm 3 leverages the UNSAT core \( \mathcal{S}_C \) to update the backtracking level \( dl \). Since \( \mathcal{S}_C \) contains both constraints that encode the program semantics and constraints from the invariant candidate \( I \), by intersecting \( I \) with \( \mathcal{S}_C \) (Line 13), we are able to extract the conflict predicate \( \delta_i \) that falsifies \( F_I \).

We leverage the conflict predicate \( \delta_i \) to prune the search space, by forcing the baseline DFS traversal procedure to perform a non-chronological backtracking. Technically, this is accomplished by changing the value of the backtracking level (\( dl \)), which is a global variable. This allows our method to skip any redundant invariant candidates that share the same conflict predicate \( \delta_i \).

For the running example in Figure 3, without the help of \( \delta_C \), the baseline DFS traversal would have changed the value of Node 7 of AST to the \( \lor \) type and obtain AST+1, shown on the left of Figure 5 (b). Unfortunately, since the new invariant candidate still contains \( \delta_C \), it would fail verification again. Furthermore, if the DFS traversal continues along this subtree, it may generate many other ASTs, all of which contain \( \delta_C \) and thus would fail for the exact same reason.

In contrast, our LR based optimization would force the DFS traversal to backtrack to Node 5 of the current AST, by changing \( N_{dl} \) to 5 as shown on the right of Figure 5 (a). As a result, it would avoid generating the large number of redundant ASTs. Instead, the new AST+1 would be the one shown on the right of Figure 5 (b), where the conflict predicate \( k = k \) is now replaced by \( k = k + 5 \).

As shown in Algorithm 3, with the conflict predicate \( \delta_i \), our method conducts two types of optimizations: clause memorization and non-chronological backtracking.

For clause memorization, we compute a forbidden set, \( C \), which is the union of all conflict predicate sets (\( \delta_i \)). To avoid growing the forbidden set infinitely, we bound the size of \( C \) to a constant by removing the less frequently used predicate, following the popular least recently used (LRU) policy for cache replacement. In this context, however, the frequency refers to the number of invariant candidates that have conflicts with the predicate.

For non-chronological backtracking, we compute a map \( M_I \) which, given a node index, returns the corresponding subtree of the invariant candidate \( I \). Here, \( M_I \subseteq AST \) means the AST representing \( \delta_i \) is a subtree of the AST representing \( I \). Using \( M_I \), we can locate the node \( n_c \) corresponding to the conflict predicate \( \delta_i \), as shown in Line 16 of Algorithm 3. Based on node \( n_c \), we can modify the backtracking level \( dl \) accordingly.

4.3 The Strengthening Predicate

It is worth noting that, if the mirror formula \( \neg M_F \) is SAT, it does not imply the validity or invalidity of \( I \). Furthermore, there is no conflict predicate that falsifies \( F_I \). Although \( I \) does not yield conflicts in this case, it still fails the inductive part of verification. The reason is that \( I (\nu) \) is not strong enough to imply \( I (\nu') \). In other words, the failure is due to the inherent weakness of \( I \), rather than the conflict predicate of \( I \). In such a case, we try to strengthen \( I \) to make it inductive (Lines 6-10 of Algorithm 3).

In general, there can be two reasons why a candidate fails the verification. One reason is that it is overly constrained, e.g., by a conflict predicate, and the other reason is that it is under constrained. In the latter case, we try to strengthen it by conjoining with an additional predicate.

In the running example, for the invariant represented by \( I_{i+1} \) on the right of Figure 5 (b), the strengthening predicate would be \( \delta_S = (i = \bar{i}) \). The conjoined formula \( I_{i+1} \land (i = \bar{i}) \) is able to pass the check \( F_I \).

This is known as abductive reasoning in the literature [18–20, 53], and such techniques have been implemented in many existing tools. Our method relies on the built-in get-abduct function of the CVC5 solver to implement a subroutine named ObtainAbduct-Pred, which starts with a true but not inductive invariant \( I \), and iteratively strengthens it.

In Lines 6-10 of Algorithm 3, we invoke the subroutine when the current candidate \( I \) is consistent with the program semantics but not yet inductive. It is worth noting that not all solutions returned by CVC5 are feasible and useful. That is why, in Lines 8 and 10, we check the feasibility of \( \delta_S \) and make sure it can make \( I \) inductive. The inductive candidate \( I_i \) (Line 10) is subsequently used for further lightweight checkings similar to Sec 3.4.1.

5 RL BASED PRIORITIZATION

In this section, we present our RL based optimization implemented in the subroutine Prioritize_by_RL, which is used by Algorithm 1. At this moment, the UNSAT core \( \mathcal{S}_C \), the conflict predicate \( \delta_C \), and the roll out trace \( T \) have all been computed for the failed candidate \( I \). The roll out trace \( T \), in particular, represents a sequence of values chosen during the construction of \( I \). Internally, our method first computes the available information to compute the reward, and then relies on the reinforcement learning (RL) agent to compute a policy gradient. Finally, the policy gradient will be used to update the data structure \( P_{RL} \) used by Algorithm 2.

In the remainder of this section, we present our method in detail.

5.1 The Policy \( P_{RL} \)

Inside Algorithm 2, if there are multiple values that can be used to fill the current node \( N_{dl} \), the invariant synthesizer picks a value for \( N_{dl} \) based on a probability distribution of these values provided by \( P_{RL} \). For instance, if the type of the current node \( N_{dl} \) is \( \chi_{dl} = \phi \), which may have values \( \neg \), \( \lor \) and \( \land \), and if the probabilities for these values are 0.12, 0.16 and 0.72, respectively, the likelihood of picking the operator \( \land \) will be the highest.

Our RL based optimization ensures that \( P_{RL} \) represents a policy that maximizes the chance of generating good invariants. Toward this end, we model the search for invariants in the hypothesis space as a Markov Decision Process (MDP), where a state is represented by the partial invariant \( I \) together with \( N_{dl} \), the node whose value will be filled next, and an action represents a possible value for \( N_{dl} \).

We use standard reinforcement learning techniques over the MDP to compute the policy \( P_{RL} \), which is then represented as a GRU network shown in Figure 6. \( P_{RL} \) takes a state \( <I, N_{dl}> \) as input, and outputs the probability of each syntactic construct associated with \( N_{dl} \), such as negation \( \neg \) and conjunction \( \land \).
with negative rewards. Furthermore, we extract more candidates from a failed candidate and use the derived candidates to provide fine-grained feedback to the RL agent. In contrast, prior works such as Chen et al. [15] only give negative rewards to the failed candidates. Their sparse reward design makes it more difficult for reinforcement learning to converge.

5.3 Generating More Feedback

To amplify the feedback from a failed candidate, we propose techniques for deriving other bad candidates \( I' \) from a bad candidate \( I \), such that \( I' \) fails the verification for a similar reason. In other words, we can use \( I' \) to update the policy without exploring it in the first place.

Recall that in Section 4, we compute the \( \text{UNSAT} \) core \( S_C \) for a bad candidate \( I \), together with the set of conflict predicates \( \delta_c \), which is a subset of the constraints of \( I \). Taking \( S_C \) and \( \delta_c \) as input, we obtain \( I' \) by mutating operators or operands in \( I \) such that \( I' \land P \) (i.e., \( I' \land (S_C \land \delta_c) \)) remains \( \text{UNSAT} \). It ensures that \( I' \) fails verification due to the same \( \text{UNSAT} \) Core.

As an example, consider the failed \( I = \{ x = \overline{x} \land k = \overline{k} \} \) in the motivating example, from which we can obtain the \( \text{UNSAT} \) Core \( S_C = \{ kN = k + 1 \land \overline{kN} = \overline{k} + 5 \land k = \overline{k} \land kN = \overline{kN} \} \) and the conflict predicate \( \delta_c = \{ k = \overline{k} \land kN = \overline{kN} \} \).

Assume that the difference between the two, \( S_C \land \delta_c = \{ kN = k + 1 \land \overline{kN} = \overline{k} + 5 \} \), encodes part of the program semantics \( P \). In this case, we may mutate \( I \) to obtain \( I' = \{ x = \overline{x} \land k + 1 = \overline{k} \} \). Since \( I' \land P \) remains \( \text{UNSAT} \), the newly created \( I' \) is guaranteed to fail verification for the same reason.

Given the reward function, policy gradient methods [66] can be used to update the policy \( \mathcal{P}_{rl} \). Recall that, in Algorithm 2, each invariant candidate \( I \) corresponds a rollout trajectory \( T = \{ (s_1, a_1, r_1), (s_2, a_2, r_2), ..., (s_{|T|}, a_{|T|}, r_{|T|}) \} \), which is a sequence of state-action-reward tuples, obtained by picking the actions using the current policy \( \mathcal{P}_{rl} \). In the final state, \( s_{|T|} \Rightarrow \overline{I, NJ} \). Each candidate \( I \) corresponds to a trace \( T \). A set of new traces \( T' \) is obtained by the newly generated \( I' \). To amplify the solver feedback, the policy gradient is computed based on a set of traces \( T' \) rather than a single trace \( T \). The objective of policy gradient methods is to update the policy \( \mathcal{P}_{rl} \) such that it maximizes the expected cumulative reward.

6 EVALUATION

We have implemented our method in a software tool (Code2RelInv), which relies on LLVM 3.6 to parse the merged C programs and construct the internal representation (IR) for the programs. It considers three types of relational properties: equivalence, continuity and non-interference, by encoding them uniformly at the IR level as a set of logical constraints. For equivalence, the encoding (e.g., \( x = \overline{x} \)) is straightforward. For continuity, the encoding is guided by the set of continuity analysis rules from Chaudhuri et al. [12]. For non-interference, it adopts the instrumentation-based technique of Chen et al. [13] to account for secret-induced resource usage.

Our baseline SyGuS search procedure is implemented in C++. Our LR based optimization is implemented using Z3 as the SMT solver to compute conflict predicates. It also uses CVC5 to compute abductive predicates. Our LR based optimization is implemented...
We ran all experiments on Cloudlab with Intel Xeon Silver CPU 2.20GHz along with NVIDIA 12GB PCI P100 GPU.

To answer RQ.2, we compared the running time of our method, with and without the learning based optimizations, also in Table 2. The running time of the baseline with syntactic filtering ($T_{\text{base}}$) is the largest, including three T/O cases (E4, E7 and E8). The reason why E4, E7, and E8 are difficult is because the invariants needed to prove these properties are more complex and the depth of their corresponding ASTs are 5-8. As a result, the baseline version has to explore an extremely large candidate space. With the RL based optimization, the running time ($T_{\text{RL}}$) is significantly reduced; all benchmarks are completed within 0.5 hour. With both the RL and the LR based optimizations, the running time ($T_{\text{RL+LR}}$) becomes the shortest.

To better understand why our RL and LR based optimizations are effective, we also collected the number of invariant candidates explored by our method. The results are shown in Table 3. Here, $n_{\text{base}}$ is the number of ASTs (of invariant candidates) explored by the baseline SyGuS search. $n_{\text{RL}}$ is the number of ASTs explored after adding RL-based optimization, and $n_{\text{RL+LR}}$ is the number of ASTs explored after adding both optimizations. For each benchmark, the minimal number is in bold font.

The results show that, among the three versions, $n_{\text{RL+LR}}$ is always the smallest. Furthermore, in many cases, such as E1, the reduction is drastic (from 1389 candidates to 5 candidates). On average, our method is able to skip $\geq 89.4\%$ of invariant candidates.

We also investigated the two individual components in LR based pruning, for computing conflict predicates and abductive predicates, respectively. We found that, in general, the time to compute conflict predicates is short and yet non-chronological backtracking based on these conflict predicates is almost always effective in speeding up our method. In contrast, the time to compute abductive predicates may be significantly longer, and may not always speed up our method. In E4, E7, C1, C3 and C5, it took an extremely long time. This is because the get-abduct function of CVCS, which is the abductive reasoning routine used in our method, may diverge in an infinite chain of speculations. Thus, unlike conflict predicates, abductive predicates must be used more judiciously.

Nevertheless, our results show that, by using abductive predicates and conflict predicates in the same procedure, we can improve the overall performance consistently.

### Related Work

**Invariant Synthesis.** We have already mentioned two closely related invariant synthesis techniques: Code2Inv [64] and LinearArbitrary [72]. LinearArbitrary is a data-driven technique, which uses sampled data to generate linear classifiers. Other examples in this category include ICE-DT [27, 28], LoopInvGen [48, 49], Guess-and-Check [60, 61], and [29, 58]. A problem with these techniques is that, while the synthesized predicates are consistent with sampled data, they may over-fit and thus produce unnecessarily
Table 2: Comparing the performance of our method (Code2RelInv) and the existing method Code2Inv [64].

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Code2RelInv</th>
<th>Code2Inv [64]</th>
<th>Code2RelInv</th>
<th>Code2Inv [64]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalence</td>
<td>(t_b)</td>
<td>(t_R)</td>
<td>(t_{R+L})</td>
<td>(t_{R+L+L-R})</td>
</tr>
<tr>
<td>E1</td>
<td>1389</td>
<td>147</td>
<td>5</td>
<td>✓</td>
</tr>
<tr>
<td>C2</td>
<td>1627</td>
<td>139</td>
<td>22</td>
<td>✓</td>
</tr>
<tr>
<td>E3</td>
<td>1395</td>
<td>245</td>
<td>18</td>
<td>✓</td>
</tr>
<tr>
<td>E4</td>
<td>8735</td>
<td>4086</td>
<td>27</td>
<td>✓</td>
</tr>
<tr>
<td>E5</td>
<td>1538</td>
<td>175</td>
<td>35</td>
<td>✓</td>
</tr>
<tr>
<td>E6</td>
<td>921</td>
<td>131</td>
<td>13</td>
<td>✓</td>
</tr>
<tr>
<td>E7</td>
<td>2612</td>
<td>172</td>
<td>7</td>
<td>✓</td>
</tr>
<tr>
<td>E8</td>
<td>10483</td>
<td>7241</td>
<td>37</td>
<td>✓</td>
</tr>
<tr>
<td>Continuity</td>
<td>(t_b)</td>
<td>(t_R)</td>
<td>(t_{R+L})</td>
<td>(t_{R+L+L-R})</td>
</tr>
<tr>
<td>C1</td>
<td>1402</td>
<td>397</td>
<td>114</td>
<td>✓</td>
</tr>
<tr>
<td>C2</td>
<td>1360</td>
<td>445</td>
<td>85</td>
<td>✓</td>
</tr>
<tr>
<td>C3</td>
<td>1397</td>
<td>352</td>
<td>123</td>
<td>✓</td>
</tr>
<tr>
<td>C4</td>
<td>1528</td>
<td>392</td>
<td>216</td>
<td>✓</td>
</tr>
<tr>
<td>C5</td>
<td>1245</td>
<td>461</td>
<td>176</td>
<td>✓</td>
</tr>
<tr>
<td>C6</td>
<td>1169</td>
<td>417</td>
<td>183</td>
<td>✓</td>
</tr>
<tr>
<td>C7</td>
<td>1443</td>
<td>305</td>
<td>161</td>
<td>✓</td>
</tr>
<tr>
<td>E1</td>
<td>46.20</td>
<td>33.88</td>
<td>9.83</td>
<td>✓</td>
</tr>
<tr>
<td>E2</td>
<td>49.87</td>
<td>29.01</td>
<td>15.33</td>
<td>✓</td>
</tr>
<tr>
<td>E3</td>
<td>49.26</td>
<td>40.41</td>
<td>14.12</td>
<td>✓</td>
</tr>
<tr>
<td>E4</td>
<td>T/O</td>
<td>1225.41</td>
<td>809.65</td>
<td>✓</td>
</tr>
<tr>
<td>E5</td>
<td>42.35</td>
<td>37.49</td>
<td>17.29</td>
<td>✓</td>
</tr>
<tr>
<td>E6</td>
<td>48.25</td>
<td>38.36</td>
<td>15.14</td>
<td>✓</td>
</tr>
<tr>
<td>E7</td>
<td>T/O</td>
<td>218.37</td>
<td>24.50</td>
<td>✓</td>
</tr>
<tr>
<td>E8</td>
<td>T/O</td>
<td>1642.55</td>
<td>1140.72</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 3: Comparing the number of invariant candidates explored by our method with different optimizations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Inv. Candidates</th>
<th>Benchmark</th>
<th>Inv. Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalence</td>
<td>(N_b)</td>
<td>(N_R)</td>
<td>(N_{R+L})</td>
</tr>
<tr>
<td>E1</td>
<td>1389</td>
<td>147</td>
<td>5</td>
</tr>
<tr>
<td>C2</td>
<td>1627</td>
<td>139</td>
<td>22</td>
</tr>
<tr>
<td>E3</td>
<td>1395</td>
<td>245</td>
<td>18</td>
</tr>
<tr>
<td>E4</td>
<td>8735</td>
<td>4086</td>
<td>27</td>
</tr>
<tr>
<td>E5</td>
<td>1538</td>
<td>175</td>
<td>35</td>
</tr>
<tr>
<td>E6</td>
<td>921</td>
<td>131</td>
<td>13</td>
</tr>
<tr>
<td>E7</td>
<td>2612</td>
<td>172</td>
<td>7</td>
</tr>
<tr>
<td>E8</td>
<td>10483</td>
<td>7241</td>
<td>37</td>
</tr>
<tr>
<td>Continuity</td>
<td>(N_b)</td>
<td>(N_R)</td>
<td>(N_{R+L})</td>
</tr>
<tr>
<td>C1</td>
<td>1402</td>
<td>397</td>
<td>114</td>
</tr>
<tr>
<td>C2</td>
<td>1360</td>
<td>445</td>
<td>85</td>
</tr>
<tr>
<td>C3</td>
<td>1397</td>
<td>352</td>
<td>123</td>
</tr>
<tr>
<td>C4</td>
<td>1528</td>
<td>392</td>
<td>216</td>
</tr>
<tr>
<td>C5</td>
<td>1245</td>
<td>461</td>
<td>176</td>
</tr>
<tr>
<td>C6</td>
<td>1169</td>
<td>417</td>
<td>183</td>
</tr>
<tr>
<td>C7</td>
<td>1443</td>
<td>305</td>
<td>161</td>
</tr>
</tbody>
</table>

complicated invariants. We have shown an example of this problem in Section 2. Our method does not have this problem, because it focuses on the program semantics instead of the sampled data.

**Code2Inv** [64] is a *neural network based technique*, which utilizes graph neural networks to encode the program dependency and TreeLSTM to embed the partial invariant [63]. Other techniques in this category include Cl2Inv [57] and G-CLN [71]. The main problem with these techniques is that, since neural networks focus on encoding program dependency information, they are often ineffective in synthesizing *relational* predicates. This has been confirmed by our experiments in Section 6. There are also other techniques for synthesizing *polar invariants* using program analysis techniques such as symbolic execution [44–46], abstract interpretation [54, 55], or compositional recurrence analysis [21, 36, 37]. However, they all target a single program, whereas our method aims to verify relational properties.

There is also a class of constrained Horn clause (CHC) solvers, developed for generating loop invariants but in principle may be used to verify relational properties as well. We have evaluated a state-of-the-art CHC solver, SpaceC [38]. Unfortunately, it returns *unknown* for most of our benchmarks. Given a set of CHC constraints with unknown predicate symbols, the CHC solver aims to produce a definition of the unknown predicate symbols such that all the constraints are satisfied. This is accomplished by first checking if all bounded unrollings of the CHC system satisfy the constraints, and then increasing the bound gradually until the proof no longer depends on the bound. Some CHC solvers [3, 35, 42, 56, 72] focus on developing new unwinding techniques while other solvers [34, 38, 43, 62] implicitly unwind the system. Specifically, Shemer et al. [62] refine the property directed inference technique to support relational verification, which is orthogonal to our learning-based method for producing relation invariant.

**Program Synthesis.** Besides invariant synthesis, learning based techniques have been used to improve program synthesis [10, 40, 41, 63]. Most of them utilize on-policy learning and often take the verifier’s result as is. An exception is Chen et al. [15] who perform off-policy learning and incorporate some additional feedback from the verifier. However, it does not use fine-grained feedback such as the ones computed by our method, from both the conflict predicates and the abductive predicates. Furthermore, the enumerative search procedure in [15] may produce ill-formed candidates, which do not occur in our method.

Besides learning, other types of information have also been used for pruning the search space [22–24, 31, 52, 68]. Some of them leverage semantic information of the DSL to check the feasibility of partial programs [22, 23], while others, such as BlAZE [68], use abstract interpretation to build the space of feasible programs. There are also type-directed pruning techniques to avoid infeasible programs [24, 26, 31, 47, 52]. However, our LR based pruning goes far beyond by pruning these well-formed but semantically-weak program candidates.

**Relational Verification.** In relational verification, one widely used approach is to carefully craft a domain-specific proof logic [9, 13, 65, 70] or a set of domain-specific proof rules [12]. Another approach is to construct and leverage a merged program via syntactic or semantic alignment [7, 14, 16, 51, 62]. While the two approaches differ, both require high-quality *relational invariants* to make the proof go through. While some prior works in this domain [13, 69] also involve invariant synthesis, they focus on simple equalities
which are too weak for most of benchmarks used in this paper, including the motivating example in Section 2. Furthermore, unlike our method, which requires the invariants to be both inductive and sufficient, they do not guarantee that the generated invariants are sufficient [13].

8 CONCLUSION
We have presented a method for synthesizing relational invariants that are guaranteed to be both inductive and sufficient. Our method leverages both syntax-guided synthesis (SyGuS) and learning-based techniques to prune the search space and prioritize the search. We have evaluated our method on a diverse set of relational verification benchmarks where the properties include equivalence, continuity, and non-interference. The experimental results show that our method can generate high-quality invariants for all cases whereas a state-of-the-art invariant synthesis tool fails most of the time. Furthermore, our learning-based optimizations drastically reduce the search space.

REFERENCES


