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:
Jingbo Wang and Chao Wang. 2022. Learning to Synthesize Relational Invariants. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556942

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, P_1 and P_2, and created a merged program P that executes instructions from P_1 and P_2 in lockstep; then we specified the equivalence relation as $\{\Phi\}P\{\Psi\}$, which is a Hoare triple [33] saying that, if P_1 and P_2 start from the same state (Φ), after the lockstep execution, they must end at the same state (Ψ). 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 $\varphi = \langle\Phi, \Psi\rangle$, where Φ is the precondition and Ψ is the postcondition. The output of Code2RelInv, which is a relational invariant $\mathcal{I}$, 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

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3556942

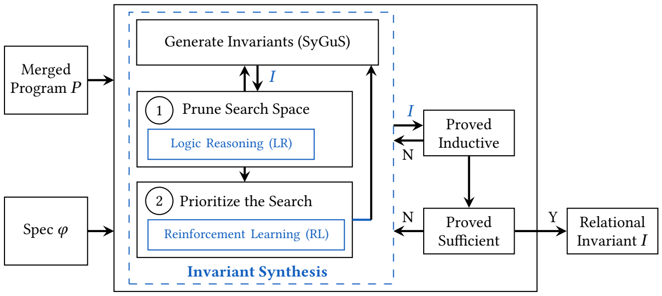


Figure 1: Code2RelInv: Our invariant synthesis method.

(a)
```
int P1(int x,int n)
int i , k = 0;
for (i=0; i!=n; ++i)
    x += k*5;
    k += 1;
    if (i >= 5)
        k += 3;


return x;
```

(b)
```
int P2(int x̄,int n̄)
int ī, k̄ = 0;
for( ī=0; ī!=n̄; ++ī)
    x̄ += k̄;
    k̄ += 5;
    if (ī >= 5)
        k̄ += 15;


return x̄;
```

(c)
```
Φ : { x = x̄, n = n̄}
int P′(int x,int n,int x̄,int n̄)
int i , k = 0; int ī, k̄ = 0;
while (i!=n && ī != n̄)
    x += k*5; x̄ += k̄;
    k += 1; k̄ += 5;
    if (i >= 5) k += 3;
    if (ī >= 5) k̄ += 15;
    i++; ī ++;
return x, x̄;
Ψ : { x = x̄ }
```

Figure 2: Given two programs P_1 and P_2, we merge them into a single program P to execute the instructions in lockstep.

from past mistakes. Specifically, our LR based pruning relies on the SMT solver's ability to generate unsatisfiability (UNSAT) cores. The second technique is reinforcement learning (RL) based *search prioritization*: the idea is to identify the part of the candidate space that is more promising and explore it first. This is accomplished by treating the invariant synthesis process as a Markov Decision Process (MDP) and use the verifier's results as positive and negative rewards to compute an exploration policy. Details of our LR and RL based techniques can be found in Sections 4 and 5, respectively.

While learning has been used to generate invariants before, e.g., in [64] and [72], they do not target relational invariants. The difference is important, for two reasons. First, the predicates must be *relational*, i.e., consisting of variables from different programs. By definition, these variables have no standard control/data flow dependencies in the merged program. Thus, any prior technique relying on the standard program dependencies would not work. Second, while prior techniques may check whether a generated invariant is *inductive*, they do not check whether it is *sufficient*. Thus, in many cases, it remains unclear how useful the generated invariants are in proving the property at hand.

Our method overcomes the above two challenges. At a high level, it can be understood as a way to intelligently aggregate and learn from past mistakes. Whenever the verifier declares an invariant candidate I as either not inductive or not sufficient, we use the information to avoid generating invariant candidates from the same equivalence class as I in the future. We also use the information to learn an exploration policy to identify invariant candidates that may have a higher chance of passing the verification.

We have evaluated the proposed method on a diverse set of 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 $k * c$ 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 $\overline{x}$ 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 $\varphi := \{\Phi\}P\{\Psi\}$, meaning that, from a state where the precondition Φ holds, executing P leads to a state where the postcondition Ψ holds. Since loops are the most challenging part in program verification, without loss of generality, we denote the merged program as $P :=$ **while** g **do** S. In this context, we want an invariant $\mathcal{I}$ of the program P with respect to the property φ to satisfy three conditions:

(a) the precondition Φ implies $\mathcal{I}$ at the beginning of the loop, denoted $\Phi \rightarrow \mathcal{I}$;

(b) $\mathcal{I}$ being true at the beginning of a loop implies $\mathcal{I}$ being true at the end of the loop, denoted $\{\mathcal{I} \wedge g\}\ S\ \{\mathcal{I}\}$, and

(c) $\mathcal{I}$ being true at the end of the loop implies the postcondition Ψ, denoted $\mathcal{I} \wedge \neg g \rightarrow \Psi$.

Conditions (a) and (b) imply that $\mathcal{I}$ is *inductive*, and Condition (c) implies that $\mathcal{I}$ is *sufficient* for proving the property φ.

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 $((\bar{i} <= (0-1) \mid\mid \bar{n} >= (\bar{n}+\bar{k})) \wedge (n == n \mid\mid \bar{i} <= (\bar{k}+0)))$ 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 (such as k and $\bar{k}$) do not have control/data dependencies at all, they never show up in the same predicate.

LinearArbitrary [72] produces $(x-\bar{x} \geq 0 \wedge x-\bar{x} \leq 0 \wedge (\neg(i <= 1) \vee \bar{i} < 2) \wedge (\neg(i <= 2) \vee \neg(\bar{i} \geq 3)) \wedge \dots)$ which is over-fitted 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: $\mathcal{I} := \{x = \bar{x} \wedge k * 5 = \bar{k} \wedge i = \bar{i}\}$. 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 $\mathcal{I} := \{k = \bar{k} \wedge x = \bar{x}\}$, whose abstract syntax tree (AST) is shown in Figure 3 (a) as AST_i. Here, the label $\emptyset$ means the node is not-in-use (`NULL`). For $\mathcal{I}$ to be *inductive*, the formula below must hold:

$$\mathcal{F}_I \;:=\; (\Phi \rightarrow \mathcal{I}) \;\wedge\; (\{\mathcal{I} \wedge g\}\; S\; \{\mathcal{I}\}) \tag{1}$$

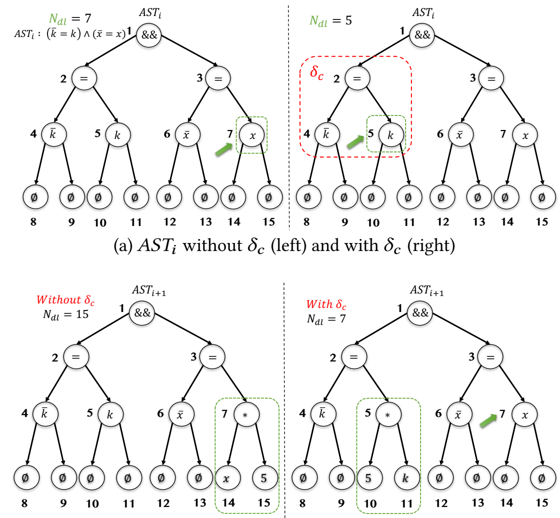
This is a classic program verification problem [25, 33], which can be solved by constructing a set of *verification conditions (VCs)* and then discharging these VCs using an SMT solver. In our method, we use Z3 [17] as the SMT solver.

For $\mathcal{I}$ to be *sufficient*, the formula below must hold:

$$\mathcal{F}_s \;:=\; (\mathcal{I} \wedge \neg g \rightarrow \Psi). \tag{2}$$

We check this formula also using the Z3 SMT solver.

Since the first invariant candidate is not inductive, it will fail the check by $\mathcal{F}_\mathcal{I}$. 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 $\mathcal{F}_\mathcal{I}$, but also fail for the same reason as the initial candidate.



(a) AST_i without δ_c (left) and with δ_c (right)

(b) AST_{i+1} without δ_c (left) and with δ_c (right)

Figure 3: Constructing the next AST by modifying the current AST. The i-th candidate shown in (a), with and without the conflict predicate δ_c. The $(i+1)$-th candidate shown in (b), with and without δ_c-based pruning.

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* $\delta_c := (\bar{k} = k)$. In other words, δ_c contradicts to the program semantics. Thus, as long as a candidate contains δ_c, it will fail to be inductive.

Since the second candidate on the left of Figure 3(b) also contains δ_c, 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 $\chi_5 = \{c * var, var\}$, 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 φ
Output: Relational invariant $\mathcal{I}$

$\mathcal{I} \leftarrow \emptyset$, $dl \leftarrow 1$, and $G \leftarrow \{(dl, \mathcal{G}) | 1 \le dl \le 2^{\mathcal{H}} - 1\}$
$C \leftarrow \emptyset$ and $\mathcal{P}_{RL} \leftarrow undef$
while running time < threshold **do**
  $\mathcal{I}, \mathcal{T} \leftarrow$ GEN_NEXT_INVCANDIDATE($\mathcal{I}, dl, G, C, \mathcal{P}_{RL}$)
  if PROVED_INDUCTIVE($P, \varphi, \mathcal{I}$) **then**
    if PROVED_SUFFICIENT($P, \varphi, \mathcal{I}$) **then return** $\mathcal{I}$
  $\mathcal{S}_C, \delta_c \leftarrow$ PRUNE_BY_LR($P, \varphi, \mathcal{I}, dl, C$) ▷ update dl, C
  PRIORITIZE_BY_RL($\mathcal{I}, \mathcal{T}, \mathcal{S}_C, \delta_c, \mathcal{P}_{RL}$) ▷ update $\mathcal{P}_{RL}$

Boolean	ϕ	:=	$p \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi$
Atomic Pred	p	:=	$a \odot a \mid \vec{a} \odot \vec{a} \mid \mathcal{A}' \odot \mathcal{A}'$
Array Expr	$\vec{a}$	:=	$\texttt{getValue}(\mathcal{A}, i) \mid \vec{a} \uplus \vec{a} \mid \lambda\vec{\mathcal{A}}.F$
Array Index	i	:=	$a \mid c$
Arith Expr	a	:=	$a_0 \mid a \uplus a \mid a \uplus c$
Arith Expr0	a_0	:=	$c * var \mid var$
Comparator	$\odot$	:=	$= \mid < \mid \le \mid > \mid \ge \mid \neq$
Operator	$\uplus$	:=	$+ \mid -$
Array Func	F	:=	$\texttt{sum}(\mathcal{A}, i_l, i_h) \mid \texttt{min}(\mathcal{A}, i_l, i_h) \mid \texttt{max}(\mathcal{A}, i_l, i_h)$
Array $\mathcal{A}'$:= $\mathcal{A} \mid \texttt{getSubset}(\mathcal{A}, i_l, i_h) \mid \texttt{getSubset1}(\mathcal{A}, i \odot c)$			

Figure 4: The DSL for relational invariants, where c is a set of constants, var is a set of variables, and $\mathcal{A}$ 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 $\varphi = \langle \Phi, \Psi \rangle$ as input, and returns the invariant $\mathcal{I}$ as output. It first initializes the data structures: $\mathcal{I}$, dl, G, $\mathcal{P}_{RL}$ and C. Here, $\mathcal{I}$ is the AST of the invariant candidate, which is initialized to NULL. The *decision level*, dl, is the index of the AST node in $\mathcal{I}$ that will be modified to generate the next invariant candidate. While modifying the AST, we follow the depth-first-search (DFS) order. Therefore, dl refers to the backtracking point during DFS. G is a data structure that maps each backtracking point dl to its `unvisited` grammar set $\mathcal{G}$; 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 $\mathcal{I}$ in the hypothesis space defined by a domain specific language (DSL). If $\mathcal{I}$ 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 $\mathcal{G}$ of the DSL for expressing the invariants. $\mathcal{G}$ 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 *atomic predicate* are $\mathcal{G}[p] = \{a \odot a, \vec{a} \odot \vec{a}, \mathcal{A}' \odot \mathcal{A}'\}$. The DSL is designed such that invariants in the DSL can be analyzed by any SMT solver that supports the popular *linear integer arithmetic (LIA)* and *array* theories.

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

Function `getValue`($\mathcal{A}, i$) returns the i-th element of the array $\mathcal{A}$, while $\lambda\vec{\mathcal{A}}.F$ denotes applying function F to array $\mathcal{A}$, 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`($\mathcal{A}, i_l, i_h$) returns another array $\mathcal{A}[i_l, i_h]$, which has a subset of the elements. Similarly, `getSubset1`($\mathcal{A}, i \odot c$) returns a subset of the elements satisfying the condition $(i \odot c)$. For instance, `getSubset1`($\mathcal{A}, i \neq 2$) returns a new array $S = \{\mathcal{A}[i] \mid i \neq 2 \wedge 0 \le i \le |\mathcal{A}|\}$.

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

3.3 Abstract Syntax Tree (AST)

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

Each node $\mathcal{N}$ has a type $\chi_{\mathcal{N}}$, which may be var, c, $\mathcal{A}$, or any element in the set $\{\phi, p, \vec{a}, i, a, a_0, \odot, \uplus, \ldots\}$, which corresponds to the set of grammar rules in Figure 4. If the type $\chi_{\mathcal{N}}$ is var, c, or $\mathcal{A}$, the node $\mathcal{N}$ corresponds to a scalar variable in var, a constant in c, or an array in $\mathcal{A}$. Otherwise, the node corresponds to a set of production rules defined by the grammar in Figure 4.

For example, if $\chi_{\mathcal{N}} = p$, the set of production rules, $\mathcal{G}[\chi_{\mathcal{N}}]$, is $\{a \odot a, \vec{a} \odot \vec{a}, \mathcal{A}' \odot \mathcal{A}'\}$. Assuming that $\vec{a} \odot \vec{a}$ is chosen, we have $\chi_{\mathcal{N}} \leftarrow \odot(\chi_{2\mathcal{N}}, \chi_{2\mathcal{N}+1})$, meaning that the two child nodes have a type $\chi_{2\mathcal{N}} = \chi_{2\mathcal{N}+1} = \vec{a}$.

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

$$\mathcal{I} := \{ (\mathcal{N}, v) \mid 1 \le \mathcal{N} <= 2^{\mathcal{H}} - 1, v \in \mathcal{G}[\chi_{\mathcal{N}}] \} \quad (3)$$

In Figure 5, for example, we have an incomplete invariant under construction $\mathcal{I}_3 = \{(1, \&\&), (2, =), (4, var_1)\}$.

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 3, 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 $\emptyset$, meaning they are not yet part of the AST. Furthermore, assume the root node has the type ϕ, meaning it is a Boolean expression. Our method starts with Node 1. If it assigns the operator "&&" to Node 1, the tree maps to $\mathcal{I}_1 := \{(1, \&\&)\}$. According to the DSL in Figure 4, the child node types must be $\chi_2 = \chi_3 = \phi$.

Our method continues with Node 2. If it assigns the operator "=" to Node 2, the tree maps to $\mathcal{I}_2 := \{(1, \&\&), (2, =)\}$. 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

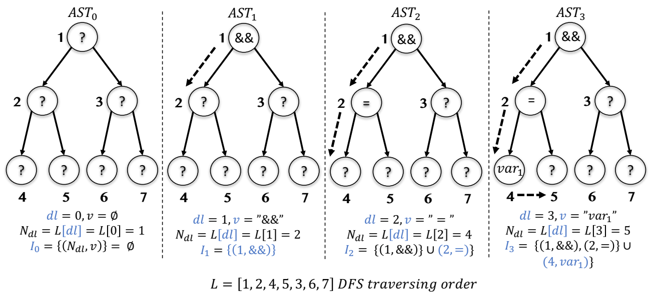


Figure 5: Step-by-step construction of invariant candidate.

candidate. Some nodes may remain $\emptyset$, meaning they are still not part of the AST.

3.3.2 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_i is the current AST, and AST_{i+1} is the next AST that our method generates. According to the DFS order, if the backtracking point ($\mathcal{N}_{dl}$) is 7, we should modify Node 7.

Since Node 7 is of the type $\chi_7 = a_0$, which may be either var or $c * var$, we change Node 7 from var to $c * var$. This results in assigning the operator "$*$" to Node 7 and then assigning values to the child nodes accordingly. The new backtracking point is set to $\mathcal{N}_{dl} = 15$.

3.4 Generating Invariant Candidates

We now present the subroutine Gen_Next_InvCandidate, 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 $\mathcal{I}_{old}$, 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 $\mathcal{G}[\chi_{dl}]$ (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 $\mathcal{G}[\chi_{dl}]$, where $\mathcal{G} = G[dl]$, the backtracking level remains unchanged. Otherwise, it becomes the last dl' where $\mathcal{G}[\chi_{dl'}]$ 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 $\mathcal{P}_{RL}$ (computed by the RL agent), and documents the history $\langle \mathcal{I}, v, 0 \rangle$ 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

Algorithm 2 Subroutine Gen_Next_InvCandidate.

1: **Input** $\mathcal{I}_{old}$, dl, G, C, and $\mathcal{P}_{RL}$.
2: **Output** $\mathcal{I}$, Trace $\mathcal{T}$
3: $\mathcal{I} \leftarrow$ values of $L[0 : dl - 1]$ in $\mathcal{I}_{old}$
4: **while** $\neg$ AllNodeAssigned(L, $\mathcal{I}$) **do**
5: $\quad\mathcal{G} \leftarrow G[dl]$; $\mathcal{N}_{dl} \leftarrow L[dl]$; $\chi_{dl} \leftarrow type[\mathcal{N}_{dl}]$
6: $\quad$**if** $\mathcal{P}_{RL} = undef$ **then**
7: $\quad\quad v \leftarrow$ Pick the first `unvisited` value in grammar $\mathcal{G}[\chi_{dl}]$
8: $\quad\quad$Label v as `visited` in $\mathcal{G}[\chi_{dl}]$ in G
9: $\quad$**else**
10: $\quad\quad v \sim \mathcal{P}_{RL}[< \mathcal{I}, \mathcal{N}_{dl} >, \chi_{dl}]$
11: $\quad$SetChildrenType($\mathcal{N}_{dl}$, v);
12: $\quad\mathcal{I} \leftarrow \mathcal{I} \cup \{(\mathcal{N}_{dl}, v)\}$; ▷ Adding to partial AST
13: $\quad\mathcal{T} \leftarrow \mathcal{T} \cup <\mathcal{I}, v, 0>$;
14: $\quad dl \leftarrow dl + 1$
15: **if** all values in $\mathcal{G}[\chi_{dl}]$ are visited **then**
16: $\quad dl' \leftarrow dl$ ▷ Backtracking
17: $\quad$**while** all values in $\mathcal{G}[\chi_{dl'}]$ are visited **do**
18: $\quad\quad dl' \leftarrow dl' - 1$;
19: $\quad\quad\mathcal{G} \leftarrow G[dl']$; $\mathcal{N}_{dl'} \leftarrow L[dl']$; $\chi_{dl'} \leftarrow type[\mathcal{N}_{dl'}]$
20: $\quad$Label $G[k]$ as `unvisited` **forall** $dl \geq k > dl'$
21: $\quad dl \leftarrow dl'$
22: **if** $\mathcal{P}_{RL} = undef$ **then**
23: $\quad dl \leftarrow 1$
24: **return** $\mathcal{I}$, $\mathcal{T}$

the obviously bad invariant candidates. First, we perform a light-weight checking of $\mathcal{I}$ before delivering it to Proved_ Inductive subroutine. For instance, with a set of conflict predicates stored in C (which are computed by our LR based optimization), we first verify if $\mathcal{I}$ is consistent with C and if it is, we reject it without further verification. Second, we enforce a lexicographical ordering over operands under commutative operators (e.g., $\wedge$), to rule out the semantically-equivalent but syntactically different invariant candidates. For instance, if $pred_1 \wedge pred_2$ has been explored before, then $pred_2 \wedge pred_1$ will not be explored in the future.

3.4.2 Verification. After generating the invariant candidate $\mathcal{I}$, we use an SMT solver based verifier to check whether $\mathcal{I}$ is inductive and sufficient, using the subroutines Proved_Inductive and Proved_Sufficient. They are based on the three conditions at the end of Section 2.1. Recall that Conditions (a) and (b) implies that $\mathcal{I}$ is *inductive* and Condition (c) implies that $\mathcal{I}$ is *sufficient*. They can be checked by first constructing two formulas, $\mathcal{F}_I$ and $\mathcal{F}_S$, based on Eq. 1 and Eq. 2 in Section 2.3, and then solving the formulas using an off-the-shelf SMT solver.

4 LR BASED PRUNING

In this section, we present our logical reasoning (LR) based optimization implemented in the subroutine Prune_by_LR, which is used by Algorithm 1. At this moment, the invariant candidate $\mathcal{I}$ has been rejected by the verifier. Our goal is to analyze the reason why $\mathcal{I}$ fails and learn from it.

Algorithm 3 shows the pseudo code. Here, the input consists of the merged program P, the relational property φ, and the failed invariant $\mathcal{I}$. The output consists of $\mathcal{S}_C$ and δ_c, where $\mathcal{S}_C$ is an unsatisfiability (`UNSAT`) core and δ_c is a conflict predicate. Together, they illustrate the reason why $\mathcal{I}$ fails the verification. Besides the 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 $\mathcal{S}_C$, computing the conflict predicate δ_c, performing non-chronological backtracking (by changing dl), and computing the strengthening predicate δ_S.

4.1 Constructing the UNSAT Core

We take the inductive part of $\mathcal{F}_\mathcal{I}$ for demonstration. $\mathcal{F}_\mathcal{I} := \forall v.\ \{\mathcal{I}(v) \wedge g(v)\}\ S(v, v')\ \ \{\mathcal{I}(v')\}$. In a loop's body S, v stands for old variables (incoming to the loop) and v' 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 as 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 $\mathcal{F}_I$ and $\mathcal{F}_S$, which are used for verification, contain universal quantifier ($\forall$), and when they fail verification, the SMT solver returns satisfying solutions for the negated formulas $\neg\mathcal{F}_I$ and $\neg\mathcal{F}_S$. 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 $\neg\mathcal{F}_I$. When it is evaluated as SAT, the solver returns a model, consisting of values assigned to the variables that makes $\mathcal{I}$ 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 Φ in the relational specification. In fact, checking if the model can be derived by the precondition Φ 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*, $\neg M_\mathcal{F}$, such that $\neg M_\mathcal{F}$ being unsatisfiable implies that $\mathcal{F}_I$ does not hold and candidate invariant $\mathcal{I}$ is invalid. Therefore, we can use the formula $\neg M_\mathcal{F}$ to extract the UNSAT core. However, it is worth noting that the reverse does not have to be true. The invalidity of $\mathcal{I}$ does not imply the unsatisfiability of $\neg M_\mathcal{F}$.

4.1.1 *The Mirror Formula.*

*Definition 1 (**Mirror Formula**).* Assuming the verification problem requires the validity of the $\mathcal{F}_I$, defined as the Hoare triple $\forall\, v.\{\mathcal{I}(v) \wedge g\}\ S(v, v')\ \{\mathcal{I}(v')\}$, the mirror formula $\neg M_\mathcal{F}$ is defined as $\neg\forall\, v.\{\mathcal{I}(v) \wedge g\}\ S(v, v')\ \{\neg\mathcal{I}(v')\}$.

Whenever $\mathcal{F}_I$ fails to be verified, we check if $\neg M_\mathcal{F}$ is unsatisfiable. If $\neg M_\mathcal{F}$ is indeed unsatisfiable, we use the UNSAT core extracted from $\neg M_\mathcal{F}$ 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 $\mathcal{F}_I$ 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** $\neg M_\mathcal{F}$). *Given an invariant candidate $\mathcal{I}$ and the corresponding $\mathcal{F}_\mathcal{I}$, the unsatisfiability of its mirror formula, $\neg M_\mathcal{F}$, implies the invalidity of $\mathcal{I}$.*

Algorithm 3 Our LR based search space pruning.

```
procedure PRUNE_BY_LR(P, φ, 𝓘)
    if CHECKUNSAT(¬M_𝓕) then
        𝓢_C ← OBTAINUNSATCORE(P, φ, 𝓘)
        δ_c ← UPDATETRAVERSEORDER(𝓢_C, 𝓘, C)
    else
        𝓢_C, δ_c ← ∅, ∅
        δ_s ← OBTAINABDUCTPRED(P, φ, 𝓘)
        if CHECKFEASIBLE(δ_s, 𝓘) then
            𝓘 ← 𝓘 ∧ δ_s
            if PROVED_INDUCTIVE(P, φ, 𝓘) then 𝓘_i ← 𝓘
    return 𝓢_C, δ_c
procedure UPDATETRAVERSEORDER(𝓢_C, 𝓘, 𝓣_NOW, C)
    δ_c ← 𝓢_C ∩ { δ_i | δ_i ∈ 𝓘 }                    ▷ conflict predicate
    C ← C ∪ δ_c
    𝓜_I ← {<n_i, δ_i> | 1 ≤ n_i ≤ 2^𝓗 − 1, δ_i ⊆_AST 𝓘 }
    n_c ← GETVALUEBYKEY(𝓜_I, δ_c)                    ▷ 𝓜_I[n_c] = δ_c
    dl ← n_c
    return δ_c
```

We provide the following formal proof to describe the the intuition behind the theorem 2, which illustrates the relationship between $\mathcal{I}$ and $\neg M_\mathcal{F}$.

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, $M_\mathcal{F} := \forall\, v.\{\mathcal{I}(v) \wedge g\}S(v, v')\{\neg\mathcal{I}(v')\}$, if $M_\mathcal{F}$ is satisfiable, then all its conjuncts, including $\neg\mathcal{I}(v')$ evaluate to true. Consequently, if $M_\mathcal{F}$ is satisfiable then $\mathcal{I}(v')$ is false, i.e., the invariant $\mathcal{I}$ is invalid. Since $M_\mathcal{F}$ is universally quantified, the solver evaluates its negated form $\neg M_\mathcal{F}$, and when it is UNSAT, it means that non-negated one is SAT, and hence, $\mathcal{I}$ is invalid. Now, UNSAT cores can be extracted from $\neg M_\mathcal{F}$ to prune the search space.

This approach catches only some cases for $\mathcal{I}$ being invalid, i.e., when $\mathcal{I}$ does not work with the fresh variables $\mathcal{I}(v')$. There could be cases when $\mathcal{F}_\mathcal{I}$ fails on its other conjuncts, e.g., on $\mathcal{I}(v)$, but the mirror formula won't be able to detect those cases. Specifically, $\mathcal{I}(v)$ may not be strong enough to imply $\mathcal{I}(v')$. 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, $\mathcal{F}_I$, 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:

$$\begin{gathered}\mathbf{a13}\text{: } kN = k + 1 \wedge\ \ \mathbf{a14}\text{: } \overline{kN} = \overline{k} + 5 \wedge \\ \mathbf{a15}\text{: } (i < 5 \wedge kNN = kN) \vee (i \geq 5 \wedge kNN = kN + 3) \\ \mathbf{a17}\text{: } (\overline{i} < 5 \wedge \overline{kNN} = \overline{kN}) \vee (\overline{i} \geq 5 \wedge \overline{kNN} = \overline{kN} + 15) \wedge \\ \mathbf{a21 : kNN} = \overline{\mathbf{kNN}}\end{gathered}$$

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 **a21** is from the invariant candidate $\mathcal{I}$, while the rest of the constraints in the UNSAT core encodes the program semantics. Therefore, our method labels **a21** as the *conflict predicate* δ_c, highlighted by the red dashed box on the right side of Figure 3(a). In other words, any invariant candidate that contains δ_c 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 $\mathcal{I}$, by intersecting $\mathcal{I}$ with $\mathcal{S}_C$ (*Line 13*), we are able to extract the *conflict* predicate δ_c that falsifies $\mathcal{F}_I$.

We leverage the conflict predicate δ_c 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 δ_c.

For the running example in Figure 3, without the help of δ_c, the baseline DFS traversal would have changed the value of Node 7 of AST_i to the * type and obtain AST_{i+1}, shown on the left of Figure 5 (b). Unfortunately, since the new invariant candidate still contains δ_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 δ_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_i, by changing $\mathcal{N}_{dl}$ 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_{i+1} would be the one shown on the right of Figure 5 (b), where the conflict predicate $\{\bar{k} = k\}$ is now replaced by $\{\bar{k} = k * 5\}$.

As shown in Algorithm 3, with the conflict predicate δ_c, 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 (δ_c). 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 $\mathcal{M}_I$ which, given a node index, returns the corresponding subtree of the invariant candidate $\mathcal{I}$. Here, $\delta_i \subseteq_{AST} \mathcal{I}$ means the AST representing δ_i is a subtree of the AST representing $\mathcal{I}$. Using $\mathcal{M}_I$, we can locate the node n_c corresponding to the conflict predicate δ_c, 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 $\neg M_{\mathcal{F}}$ is SAT, it does not imply the validity or invalidity of $\mathcal{I}$. Furthermore, there is no conflict predicate that falsifies $\mathcal{F}_I$. Although $\mathcal{I}$ does not yield conflicts in this case, it still fails the inductive part of verification. The reason is that $\mathcal{I}(v)$ is not strong enough to imply $\mathcal{I}(v')$. In other words, the failure is due to the inherent weakness of $\mathcal{I}$, rather than the conflict predicate of $\mathcal{I}$. In such a case, we try to strengthen $\mathcal{I}$ 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 $\mathcal{I}_{i+1}$ on the right of Figure 5 (b), the strengthening predicate would be $\delta_S = \{i = \bar{i}\}$. The conjoined formula $\mathcal{I}_{i+1} \wedge (i = \bar{i})$ is able to pass the check $\mathcal{F}_I$.

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 ObtainAbductPred, which starts with a *true* but not *inductive* invariant $\mathcal{I}$, and iteratively strengthens it.

In Lines 6-10 of Algorithm 3, we invoke the subroutine when the current candidate $\mathcal{I}$ 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 δ_s and make sure it can make $\mathcal{I}$ inductive. The inductive candidate $\mathcal{I}_i$ (*line 10*) is subsequently used for further light-weight 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 δ_c, and the rollout trace $\mathcal{T}$ have all been computed for the failed candidate $\mathcal{I}$. The rollout trace $\mathcal{T}$, in particular, represents a sequence of values chosen during the construction of $\mathcal{I}$. 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 $\mathcal{P}_{RL}$ used by Algorithm 2.

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

5.1 The Policy $\mathcal{P}_{RL}$

Inside Algorithm 2, if there are multiple values that can be used to fill the current node $\mathcal{N}_{dl}$, the invariant synthesizer picks a value for $\mathcal{N}_{dl}$ based on a probability distribution of these values provided by $\mathcal{P}_{RL}$. For instance, if the type of the current node $\mathcal{N}_{dl}$ is $\chi_{dl} = \phi$, which may have values "¬", "∨" and "∧", and if the probabilities for these values are 0.12, 0.16 and 0.47, respectively, the likelihood of picking the operator "∧" will be the highest.

Our RL based optimization ensures that $\mathcal{P}_{RL}$ 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 $\mathcal{I}$ together with $\mathcal{N}_{dl}$, the node whose value will be filled next, and an *action* represents a possible value for $\mathcal{N}_{dl}$.

We uses standard reinforcement learning techniques over the MDP to compute the policy $\mathcal{P}_{RL}$, which is then represented as a GRU network shown in Figure 6. $\mathcal{P}_{RL}$ takes a *state* $<\mathcal{I}, \mathcal{N}_{dl}>$ as input, and outputs the probability of each syntactic construct associated with $\mathcal{N}_{dl}$, such as negation "¬" and conjunction "∧".

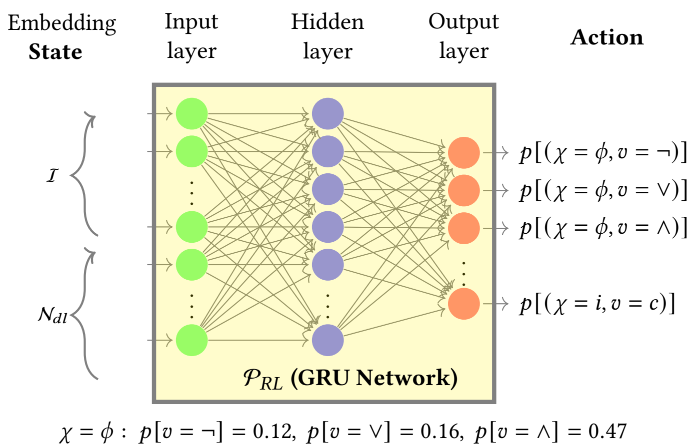


Figure 6: Illustrating the use of $\mathcal{P}_{RL}$ in our method.

5.2 The Reward Function

The main difference between our method and prior works on using RL techniques [10, 15, 63] is that, while they merely incorporate the *negative* feedback (i.e., the candidate $\mathcal{I}$ has been rejected by the verifier), we are able to extract a richer set of *positive* and *negative* feedbacks from the verifier. Thus, we can more effectively aggregate and amplify feedbacks from the verifier.

At the center of our method is the *reward function* constructed using results of our logical reasoning (LR) subroutine. It aims to penalize *bad* candidates and reward *good* candidates. Here, *bad* candidates are the ones that contradict to the semantics of the merged program P. Whenever a candidate $\mathcal{I}$ contradicts to the program semantics, we can construct an unsatisfiable formula and leverage the SMT solver to compute an UNSAT core $\mathcal{S}_C$. In contrast, *good* candidates are the ones that are consistent with the program semantics. Recall that whenever $\mathcal{I}$ is consistent with the program semantics, it satisfies the mirror formula $\neg\mathcal{M}_{\mathcal{F}}$, although $\mathcal{I}$ is still not *inductive*.

Based on the observation, we define the reward function

$$r(<\mathcal{I}, \mathcal{N}>) = \begin{cases} 0 & \text{if } \mathcal{I} \text{ is incomplete} \\ \text{o/w} & \begin{cases} +1 & \text{if } \neg\mathcal{F}_{\mathcal{I}} \text{ is UNSAT} \\ \text{o/w} & \begin{cases} +0.5 & \text{if } \mathcal{S}_C = \emptyset \quad (i.e., \neg M_{\mathcal{F}}) \text{ is SAT)} \\ -1 & \text{if } \mathcal{S}_C \neq \emptyset \quad (i.e., \neg M_{\mathcal{F}}) \text{ is UNSAT)} \end{cases} \end{cases} \end{cases} \tag{4}$$

The function assigns the reward only when $\mathcal{I}$ is a complete AST, i.e., each AST node has been assigned either a concrete value or NULL. If $\mathcal{I}$ is a good candidate (i.e., $\neg\mathcal{F}_I$ is unsatisfiable), it assigns +1 as the reward. Otherwise, in the second *o/w* case of Eq. (4), we check if $\neg M_{\mathcal{F}}$ is satisfiable (i.e., $\mathcal{S}_C = \emptyset$), which means $\mathcal{I}$ is consistent with the program semantics but not yet inductive. Thus, we assign $\mathcal{I}$ a positive reward of +0.5, to bias the exploration toward this direction. However, if $\neg M_{\mathcal{F}}$ is unsatisfiable (i.e., $\mathcal{S}_C \neq \emptyset$), based on Theorem 2, predicates in $\mathcal{I}$ must contradict to the program semantics. Thus, we assign $\mathcal{I}$ a negative reward of -1, to bias the exploration against this direction.

Thus, our design aims to provide potentially valid (*good*) candidates with positive rewards and always-conflicting (*bad*) candidates 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 ($\mathcal{I}'$) from a bad candidate $\mathcal{I}$, such that $\mathcal{I}'$ fails the verification for a similar reason. In other words, we can use $\mathcal{I}'$ to update the policy without exploring it in the first place.

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

As an example, consider the failed $\mathcal{I} = \{x = \overline{x} \wedge k = \overline{k}\}$ in the motivating example, from which we can obtain the UNSAT Core $\mathcal{S}_C = \{kN = k + 1 \ \wedge \overline{kN} = \overline{k} + 5 \ \wedge \mathbf{k} = \overline{\mathbf{k}} \wedge \mathbf{kN} = \overline{\mathbf{kN}}\}$ and the conflict predicate $\delta_c = \{\mathbf{k} = \overline{\mathbf{k}} \wedge \mathbf{kN} = \overline{\mathbf{kN}}\}$.

Assume that the difference between the two, $\mathcal{S}_C \setminus \delta_c = \{kN = k + 1 \ \wedge \overline{kN} = \overline{k} + 5\}$, encodes part of the program semantics P. In this case, we may mutate $\mathcal{I}$ to obtain $\mathcal{I}' = \{x = \overline{x} \wedge k + 1 = \overline{k}\}$. Since $\mathcal{I}' \wedge P$ remains UNSAT, the newly created $\mathcal{I}'$ 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 $\mathcal{I}$ corresponds a rollout trajectory $\mathcal{T} = \{(s_1, a_1, r_1), (s_2, a_2, r_2) \ldots (s_{|\mathcal{T}|}, a_{|\mathcal{T}|}, r_{|\mathcal{T}|})\}$, 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, $s_{|\mathcal{T}|} = <\mathcal{I}, \mathcal{N}_{dl}>$. Each candidate $\mathcal{I}$ corresponds to a trace $\mathcal{T}$. A set of new traces $\mathcal{T}'$ is obtained by the newly generated $\mathcal{I}'$. To *amplify* the solver feedback, the policy gradient is computed based on a set of traces $\mathcal{T}'$ rather than a single trace $\mathcal{T}$. 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

Table 1: The list of relational verification benchmarks.

Name	Description of the verification problem	Name	Description of the verification problem
E1	Strength Reduction [7]	N1	Array Safe [4]
E2	Loop Simple Optimization [7]	N2	Array Unsafe [4]
E3	Loop Align [7]	N3	LoopAndBranchSafe [4]
E4	Loop Pipelining [7]	N4	LoopAndBranchUnSafe [4]
E5	Loop Sinking [7]	N5	NoSecret Safe [4]
E6	Loop Unswitching [7]	N6	NoTaint Unsafe [4]
E7	Loop Var Reduction [7]	N7	Sanity Safe [4]
E8	Static Caching [7]	N8	Sanity Unsafe [4]
		N11	modPow1 Safe (DARPA STAC)
C1	BubbleSort [12]	N12	modPow1 Unsafe (DARPA STAC)
C2	Insertion-sort (Inner) [12]	N13	modPow2 Safe (DARPA STAC)
C3	Insertion-sort (Outer) [12]	N14	modPow2 Unsafe (DARPA STAC)
C4	Selection-sort (Inner) [12]	N15	k96 Safe [30, 50]
C5	Selection-sort (Outer) [12]	N16	k96 Unsafe [30, 50]
C6	Bellman-Ford [12]	N17	gpt14 Safe [30, 50]
C7	Floyd-Warshall [12]	N18	gpt14 Unsafe [30, 50]

using PyTorch, which has RL agents for computing the exploration policy. For evaluation purposes, we have compared our method with the state-of-the-art tool Code2Inv [64] on all benchmarks.

6.1 Benchmarks

Our benchmarks consist of 33 relational verification problems from three sources, as shown in Table 1. The equivalence verification problems, *E1* to *E8*, are from Barthe et al. [7]. The goal is to prove the correctness of various types of loop optimizations [1, 5, 8, 39]. The *continuity* verification problems, *C1* to *C7*, are from Chaudhuri et al.[12]. The non-interference verification problems, *N1* to *N18*, are from the DARPA STAC program and other side-channel security examples [30, 50]. Here we omit *N9, N10* since they are straight-line code without loops. While some of the programs were in Java [4], we translated them to C before applying our tool. Our underlying program verifier supports linear integer arithmetic (LIA) and array theories.

Our experiments were designed to answer two questions:

RQ.1 How effective is Code2RelInv in generating the desired relational invariants?

RQ.2 How effective are the new LR-based and RL-based techniques in reducing the search space?

We ran all experiments on Cloudlab with Intel Xeon Silver CPU 2.20GHz along with NVIDIA 12GB PCI P100 GPU.

6.2 Results for Evaluating the Effectiveness

To answer RQ.1, we applied our method to all benchmarks, and compared it with Code2Inv [64]. The results are shown in Table 2. For each benchmark, Column 1 shows the name, and Columns 2-5 show the running time of our method in seconds and the quality of the invariant. Here, T_{base} denotes the baseline, T_{+RL} denotes the baseline plus RL-based optimization, and T_{+RL+LR} denotes the baseline plus RL- and LR-based optimizations. Columns 6-7 show the running time of Code2Inv [64] and the quality of its invariant. Here, T/O means timed out after 4 hours, ✓ means the invariant is both inductive and sufficient, ✗ means the invariant is not sufficient, and – means the method fails to generate any invariant (due to assertion failure or exception).

The results in Table 2 show that our method is significantly more effective in generating relational invariants. In fact, it succeeds on all benchmarks, whereas Code2Inv [64] only succeeds on four of them. In the four cases where it succeeds, it runs more than 1000x slower than our method. It also has 4 `T/O` cases.

6.3 Results for Evaluating the Optimizations

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_{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_{+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_{+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, $\#_{base}$ is the number of ASTs (of invariant candidates) explored by the baseline SyGuS search. $\#_{+RL}$ is the number of ASTs explored after adding RL-based optimization, and $\#_{+RL+LR}$ 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, $\#_{+RL+LR}$ 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 ≥ 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 CVC5, 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.

7 RELATED WORK

Invariant Synthesis. We have already mentioned two most closely related invariant synthesis techniques: Code2Inv [64] and LinearArbitrary [72]. LinearArbitrary is a *data-driven* technique, which uses sampled data to generate linear classifies. 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].

Benchmark	Code2RelInv T_{base} (s)	T_{+RL} (s)	T_{+RL+LR} (s)	Qual	Code2Inv [64] T_{Code2Inv}(s)	Qual	Benchmark	Code2RelInv T_{base} (s)	T_{+RL} (s)	T_{+RL+LR} (s)	Qual	Code2Inv [64] T_{Code2Inv}(s)	Qual
Equivalence							*Non-interference*						
E1	46.20	33.88	**9.83**	✓	1623.07	✓	N1	44.91	36.20	**17.44**	✓	-	-
E2	49.87	29.01	**15.33**	✓	1899.45	✓	N2	18.54	29.31	**10.19**	✓	-	-
E3	49.26	40.41	**14.12**	✓	-	-	N3	46.36	37.21	**12.72**	✓	2556.73	✓
E4	T/O	1225.41	**809.65**	✓	-	-	N4	35.08	30.60	**10.05**	✓	-	-
E5	42.35	37.49	**17.29**	✓	-	-	N5	33.08	29.13	**12.08**	✓	2177.92	✓
E6	48.25	38.36	**15.14**	✓	-	-	N6	35.19	34.29	**9.88**	✓	-	-
E7	T/O	218.37	**24.50**	✓	-	-	N7	44.47	39.92	**16.34**	✓	T/O	✗
E8	T/O	1642.55	**1140.72**	✓	-	-	N8	42.52	38.83	**15.17**	✓	-	-
Continuity							N11	46.47	32.34	**13.05**	✓	T/O	✗
C1	45.82	34.95	**15.86**	✓	-	-	N12	45.86	34.22	**13.18**	✓	-	-
C2	44.95	35.12	**13.88**	✓	-	-	N13	67.02	35.25	**15.50**	✓	T/O	✗
C3	45.27	32.02	**15.22**	✓	-	-	N14	67.52	43.34	**17.46**	✓	-	-
C4	45.04	34.85	**20.67**	✓	-	-	N15	96.62	46.22	**29.18**	✓	T/O	✗
C5	44.97	35.83	**18.97**	✓	-	-	N16	102.32	49.15	**27.44**	✓	-	-
C6	43.24	35.09	**17.61**	✓	-	-	N17	91.94	46.24	**18.54**	✓	T/O	✗
C7	45.04	32.40	**17.38**	✓	-	-	N18	145.69	47.96	**14.48**	✓	-	-

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

Benchmark	Inv. Candidates $\#_{base}$	$\#_{+RL}$	$\#_{+RL+LR}$	Benchmark	Inv. Candidates $\#_{base}$	$\#_{+RL}$	$\#_{+RL+LR}$
Equivalence				*Non-interference*			
E1	1389	147	**5**	N1	1244	284	**101**
E2	1627	139	**22**	N2	276	117	**38**
E3	1953	245	**18**	N3	1381	340	**56**
E4	-	8735	**5086**	N4	323	112	**29**
E5	1538	175	**31**	N5	148	104	**21**
E6	921	131	**124**	N6	317	159	**23**
E7	-	2612	**172**	N7	1265	379	**108**
E8	-	10483	**7241**	N8	1025	356	**93**
Continuity				N11	1362	203	**57**
C1	1402	397	**114**	N12	1395	268	**52**
C2	1360	445	**85**	N13	1686	313	**64**
C3	1397	352	**123**	N14	1601	328	**49**
C4	1528	392	**216**	N15	2857	335	**127**
C5	1245	461	**176**	N16	2914	409	**156**
C6	1169	417	**183**	N17	2673	381	**114**
C7	1443	305	**161**	N18	3225	376	**68**

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 Cln2Inv [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 *polynomial 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, Spacer [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

[1] Tarek S Abdelrahman and Robert Sawaya. 2004. Improving the structure of loop nests in scientific programs. *Comput. Syst. Sci. Eng.* 19, 1 (2004), 11–25.

[2] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *International Conference on Formal Methods in Computer-Aided Design*.

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

[4] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition instead of self-composition for proving the absence of timing channels. *ACM SIGPLAN Notices* 52, 6 (2017), 362–375.

[5] David F Bacon, Susan L Graham, and Oliver J Sharp. 1994. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)* 26, 4 (1994), 345–420.

[6] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. 2016. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 116–129.

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

[8] Gilles Barthe and César Kunz. 2008. Certificate translation in abstract interpretation. In *European Symposium on Programming*. Springer, 368–382.

[9] Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. *ACM SIGPLAN Notices* 39, 1 (2004), 14–25.

[10] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. 2018. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. In *International Conference on Learning Representations*.

[11] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 913–926.

[12] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2010. Continuity analysis of programs. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 57–70.

[13] 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*. 875–890.

[14] Jia Chen, Jiayi Wei, Yu Feng, Osbert Bastani, and Isil Dillig. 2019. Relational verification using reinforcement learning. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[15] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. 2020. Program Synthesis Using Deduction-Guided Reinforcement Learning. In *International Conference on Computer Aided Verification*. Springer, 587–610.

[16] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1027–1040.

[17] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.).

[18] Isil Dillig and Thomas Dillig. 2013. Explain: a tool for performing abductive inference. In *International Conference on Computer Aided Verification*. Springer, 684–689.

[19] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive invariant generation via abductive inference. *Acm Sigplan Notices* 48, 10 (2013), 443–456.

[20] Mnacho Echenim, Nicolas Peltier, and Yanis Sellami. 2018. A generic framework for implicate generation modulo theories. In *International Joint Conference on Automated Reasoning*. Springer, 279–294.

[21] Azadeh Farzan and Zachary Kincaid. 2015. Compositional recurrence analysis. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 57–64.

[22] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices* 53, 4 (2018), 420–435.

[23] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices* 52, 6 (2017), 422–436.

[24] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (2015), 229–239.

[25] Robert W. Floyd. 1967. Assigning Meanings to Programs. In *Proceedings of Symposium on Applied Mathematics*.

[26] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed synthesis: a type-theoretic interpretation. *ACM Sigplan Notices* 51, 1 (2016), 802–815.

[27] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. 2014. ICE: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*. Springer, 69–87.

[28] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices* 51, 1 (2016), 499–512.

[29] Timon Gehr, Dimitar Dimitrov, and Martin Vechev. 2015. Learning commutativity specifications. In *International Conference on Computer Aided Verification*. Springer, 307–323.

[30] Daniel Genkin, Itamar Pipman, and Eran Tromer. 2015. Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs. *Journal of Cryptographic Engineering* 5, 2 (2015), 95–112.

[31] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program synthesis by type-guided abstraction refinement. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28.

[32] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. 2021. Finding Invariants of Distributed Systems: It's a Small (Enough) World After All.. In *NSDI*. 115–131.

[33] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580.

[34] Kryštof Hoder and Nikolaj Bjørner. 2012. Generalized property directed reachability. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 157–171.

[35] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA horn solver. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 1–7.

[36] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. 2017. Compositional recurrence analysis revisited. *ACM SIGPLAN Notices* 52, 6 (2017), 248–262.

[37] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas Reps. 2017. Non-linear reasoning for invariant synthesis. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–33.

[38] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Formal Methods in System Design* 48, 3 (2016), 175–205.

[39] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving optimizations correct using parameterized program equivalence. *ACM Sigplan Notices* 44, 6 (2009), 327–337.

[40] Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V Le, and Ni Lao. 2018. Memory Augmented Policy Optimization for Program Synthesis and Semantic Parsing. In *NeurIPS*.

[41] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. 2019. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. *arXiv preprint arXiv:1904.12584* (2019).

[42] Kenneth L McMillan. 2014. Lazy annotation revisited. In *International Conference on Computer Aided Verification*. Springer, 243–259.

[43] Dmitry Mordvinov and Grigory Fedyukovich. 2019. Property directed inference of relational invariants. In *2019 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 152–160.

[44] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 605–615.

[45] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using dynamic analysis to discover polynomial and array invariants. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 683–693.

[46] Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. Dig: A dynamic invariant generator for polynomial and array invariants. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (2014), 1–30.

[47] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices* 50, 6 (2015), 619–630.

[48] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices* 51, 6 (2016), 42–56.

[49] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2017. Loopinvgen: A loop invariant generator based on precondition inference. *arXiv preprint arXiv:1707.02029* (2017).

[50] Corina S Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run side-channel analysis using Symbolic Execution and Max-SMT. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. IEEE, 387–400.

[51] Lauren Pick, Grigory Fedyukovich, and Aarti Gupta. 2018. Exploiting synchrony and symmetry in relational verification. In *International Conference on Computer Aided Verification*. Springer, 164–182.

[52] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538.

[53] Andrew Reynolds, Haniel Barbosa, Daniel Larraz, and Cesare Tinelli. 2020. Scalable Algorithms for Abduction via Enumerative Syntax-Guided Synthesis. In *International Joint Conference on Automated Reasoning*. Springer, 141–160.

[54] Enric Rodríguez-Carbonell and Deepak Kapur. 2004. Automatic generation of polynomial loop invariants: Algebraic foundations. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*. 266–273.

[55] Enric Rodríguez-Carbonell and Deepak Kapur. 2007. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation* 42, 4 (2007), 443–476.

[56] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. 2013. Disjunctive interpolants for Horn-clause verification. In *International Conference on Computer Aided Verification*. Springer, 347–363.

[57] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. 2019. CLN2INV: learning loop invariants with continuous logic networks. *arXiv preprint arXiv:1909.11542* (2019).

[58] Rahul Sharma and Alex Aiken. 2014. From Invariant Checking to Invariant Inference Using Randomized Search. In *International Conference on Computer Aided Verification*. Springer, 88–105.

[59] Rahul Sharma and Alex Aiken. 2016. From invariant checking to invariant inference using randomized search. *Formal Methods in System Design* 48, 3 (2016), 235–256.

[60] 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.

[61] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 391–406.

[62] Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. 2019. Property directed self composition. In *International Conference on Computer Aided Verification*. Springer, 161–179.

[63] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning loop invariants for program verification. In *Neural Information Processing Systems*.

[64] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. 2020. Code2Inv: a deep learning framework for program verification. In *International Conference on Computer Aided Verification*. Springer, 151–164.

[65] Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 57–69.

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

[67] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 264–276.

[68] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.

[69] Yuepeng Wang, Isil Dillig, Shuvendu K Lahiri, and William R Cook. 2017. Verifying equivalence of database-driven applications. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–29.

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

[71] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 106–120.

[72] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. *ACM SIGPLAN Notices* 53, 4 (2018), 707–721.