Loop Invariant Inference through SMT Solving Enhanced Reinforcement Learning

Shiwen Yu
Institute for Quantum Information & State Key Laboratory of High Performance Computing, Key Laboratory of Software Engineering for Complex Systems, College of Computer Science and Technology, National University of Defense Technology
Changsha, China
yushiwen14@nudt.edu.cn

Ting Wang
Key Laboratory of Software Engineering for Complex Systems, College of Computer Science and Technology, College of Computer Science and Technology, National University of Defense Technology
Changsha, China
tingwang@nudt.edu.cn

Ji Wang*
Institute for Quantum Information & State Key Laboratory of High Performance Computing, Key Laboratory of Software Engineering for Complex Systems, College of Computer Science and Technology, National University of Defense Technology
Changsha, China
wj@nudt.edu.cn

ABSTRACT

Inferring loop invariants is one of the most challenging problems in program verification. It is highly desired to incorporate machine learning when inferring. This paper presents a Reinforcement Learning (RL) pruning framework to infer loop invariants over a general nonlinear hypothesis space. The key idea is to synergize the RL-based pruning and SMT solving to generate candidate invariants efficiently. To address the sparse reward problem in learning, we design a novel two-dimensional reward mechanism that enables the RL pruner to recognize the capability boundary of SMT solvers and learn the pruning heuristics in a few rounds. We have implemented our approach with Z3 SMT solver in the tool called LIPuS and conducted extensive experiments over the linear and nonlinear benchmarks. Experiment results show that LIPuS can solve the most cases compared to the state-of-the-art loop invariant inference tools such as Code2Inv, ICE-DT, GSpacer, SymInfer, ImplCheck, and Eldarica. Especially, LIPuS outperforms them significantly on nonlinear benchmarks.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**; • **Computing methodologies → Reinforcement learning**.

KEYWORDS

reinforcement learning, loop invariant, program verification

ACM Reference Format:
Shiwen Yu, Ting Wang, and Ji Wang. 2023. Loop Invariant Inference through SMT Solving Enhanced Reinforcement Learning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598047

*Corresponding author.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598047

1 INTRODUCTION

Program verification is very useful to guarantee the correctness, safety, and security of software. However, it is strongly limited by the capability of handling loop statements in programs. Inferring loop invariants is the fundamental and hardest problem in program verification. The goal of loop program verification is to prove that for any program state, once all the assumed logic expressions (P,"pre-conditions") before entering the loop hold, then after the loop (**while** B **do** S) terminates, all the asserted logic expressions (Q,"post-conditions") must also hold. This goal can be formulated as: $\{P\}$ **while** B **do** S $\{Q\}$ by the Hoare triple[15].

A well-known approach is to find an inductive loop invariant. Suppose we find a loop invariant I, which 1) is always true when the program first enters the loop statement, 2) is always true during the iterations in the loop statement, and 3) if jumped out the loop, can entail the post-conditions. Then we can prove the correctness of the program inductively. However, the problem of loop invariant inference is proven undecidable, and even practical instances are challenging [15].

Currently, *guess and check* approaches have been shown as a promising direction to solve loop invariants practically. These methods infer the correct loop invariant by iteratively proposing possible invariant candidates. The candidates are then checked by an SMT solver such as Z3[1]. If candidates fail to validate the loop invariant's conditions, the SMT solver will provide counterexamples (program states which falsify one of the three requirements). The counterexamples are leveraged to propose the next candidate. *Guess and check* methods often define a hypothesis space of possible loop invariants and search in the space for the correct one.

Previous *guess and check* methods are either too dependent on program behavior[12, 27, 40] or too dependent on the syntax features of the program [16, 36]. For example, ICE method [12] treats the program's source code as a black box and only considers the counterexamples feedback by the SMT solver; Code2Inv[36] mainly

[1]https://github.com/Z3Prover/z3

focuses on utilizing the syntax feature of program's source and searching the loop invariant in a "trial and error" style. Due to limited search heuristics, they can only consider a relatively small hypothesis space with limited logic connectives, predicates, and functions. Most practical loop invariants may not be included in such a small space. Therefore, we propose to combine the advantages of the two kinds of information sources in one single framework. We find it promising to utilize Reinforcement Learning (RL) to prune the large space and use SMT solving to search for the exact solution in the pruned small space. Specifically, we first set up a general template that can express every invariant in the nonlinear hypothesis space. Then, we design an RL model to prune the general template according to self-learned heuristics from the program features. After the general template is pruned to a sub-template (a template that only expresses a subset of invariants in the large hypothesis space) of which the SMT solver is capable, we use the SMT solver to solve this template such that the solved candidate invariant can pass all counterexamples. The RL pruner considers the syntax features of the program, and the SMT solver ensures the candidate is consistent with the program behavior. Our framework is depicted in Figure 1.

The insight of our framework is that reinforcement learning does well in ultra-large-scale space search problems. It is good at utilizing soft constraints such as program syntax features to perform a probabilistic search for an area where solutions possibly exist. On the other hand, queries by SMT solving can automatically search for the exact solution in a relatively small area. It can utilize hard constraints such as those implied inside the counterexamples to generate candidate invariants passing all the counterexamples. The most crucial role of this composite strategy lies in the "sub-templates". We designed a novel two-dimensional reward mechanism enabling the RL model to learn to produce suitable sub-templates in a few rounds. This reward mechanism can not only consider the traditional good-or-bad reward in reinforcement learning but also provide instructions for the next search direction, according to a case-by-case analysis of all kinds of results from the SMT solver. Enhanced by the analysis of SMT solving, ablation (Section 5.2) shows that it alleviates the sparse reward problem encountered in previous RL studies. This reward mechanism is detailed in Section 4.4.

We summarize the contributions as follows: 1) We present an RL-based pruning framework for loop invariant inference, which adopt reinforcement learning to prune on large hypothesis space over the SMT QF-NIA theory[2] and SMT solving in the pruned small space. 2) We propose a two-dimensional reward mechanism with the feedback from SMT solving. This reward mechanism alleviates the sparse reward problem and increases the RL model's probability of generating successful sub-templates. 3) We have implemented our approach in the tool called LIPuS and conducted loop invariant inference experiments over linear and non-linear benchmarks. Experiment results show that LIPuS is better at finding general loop invariants than state-of-the-art tools: Code2Inv[36], ICE-DT[13], GCLN[41], SymInfer[27], Eldarica[16], GSpacer[20], CVC5[3], and ImplCheck[31].

[2]https://www.lri.fr/~conchon/TER/2013/2/SMTLIB2.pdf, page 58.

2 RELATED WORK

Most of traditional invariant synthesis methods are focused on white-box techniques (program syntax), which can be further classified into sub-categories: abstract-interpretation-based methods [7, 8, 19], interpolation-based methods [17, 22–25], counterexample guided abstraction refinement-based methods [1, 2, 6, 14, 31, 43], logical abduction-based methods [4, 10], symbolic execution trace-based method [27], model-checking-based method [16, 20], etc. ICE [12] is a black-box method (program behavior) that introduced the ICE learning framework for the implication counterexamples and split the loop invariant inference task into two parts: a learner and a teacher. Later, with the development of AI-for-SE[18, 29, 38], a variety of techniques focuses on the learner part: random search [33], decision tree [11, 13, 40, 42], support vector machine [21, 35], PAC (Probably Approximately Correct) learning [34], reinforcement learning [36], deep learning [32, 41], etc. Different from previous works, our work uses a combined strategy that can take advantage of program behavior and program syntax simultaneously.

Specifically, we involved the following SOTA methods in the evaluation.

ICE-DT[13] uses a decision tree as the learner part. ICE-DT asks experts to annotate program properties, and it can only consider a series of logic combinations of these manually defined properties (e.g., $x + y, 2 * x, x * y$). Therefore, ICE-DT can be very efficient on the loop invariants that can be expressed by the manually defined properties. However, when the loop invariant is beyond the scope, ICE-DT will fail to solve it. Later, ICE-DT-Interval[40] improve the performance of ICE-DT by using interval counterexamples rather than traditional concrete counterexamples.

CLN [32] and GCLN [41] are deep learning methods that build a "bijection" between logic expressions and neural networks. It infers the loop invariant by training a neural network to fit in the program traces. The corresponding logic expression of the trained neural network is then regarded as the loop invariant. So far, it can not ensure that the invariant inferred from the program trace can be adequate to verify the correctness. Also, it can not ensure the trained neural network can perfectly classify the program trace.

SymInfer [27] utilizes symbolic execution trace to obtain the concrete program data-trace, which is further sent to DIG [28]. DIG is a dynamic method which is focusing on program behavior. It is good at solving nonlinear loop invariants. However, SymInfer is not a robust loop invariant solver. Instead, it aims at inferring an non-trivial invariants for any location at the source code. As a result, the loop invariant it solves may not be able to verify the correctness of the program.

Eldarica [16] is a model-checking-based Horn clauses solver. Horn clauses form a fragment of first-order logic, modulo various background theories, in which models can be constructed effectively with the help of model checking algorithms. The problem of loop invariant can be formulated as a Horn clauses problem with 3 clauses. It is very efficient for linear loop invariant inference but also is strongly limited on nonlinear instances.

GSpacer [20] introduces explicit global guidance into the local reasoning performed by IC3-style algorithms.They have extended the SMT-IC3 paradigm with three novel rules, designed to mitigate fundamental sources of failure that stem from locality.

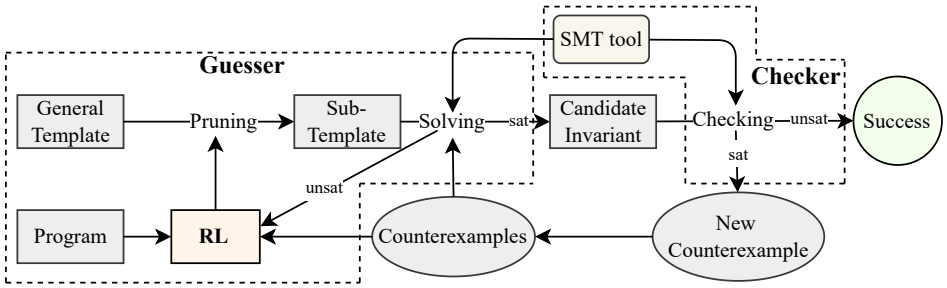

Figure 1: The framework of our method.

CVC5 [1] is an efficient open-source SMT solver. It can be used to prove the satisfiability of first-order formulas with respect to (combinations of) a variety of useful background theories. Moreover, it provides a Syntax-Guided Synthesis (SyGuS) engine to synthesize invariants.

ImplCheck [31] presents a novel SMT-based approach to synthesize implication invariants for multi-phase loops. It computes Model Based Projections to discover the program's phases, and leverages data learning to get relationships among loop variables. It is effective in mutually-dependent periodic phases, where many implication invariants need to be discovered simultaneously. Its approach has shown promising results in verifying programs with complex phase structures.

The evaluation (see Section 5) shows the effectiveness of our method compared with the above SOTA methods.

3 PRELIMINARY AND MOTIVATING EXAMPLE

In this section, we introduce the background of this work by a motivating example. After recalling the formulation of loop invariants with an example program, we use two methods mostly close to our work, namely ICE template iteration method [12] and RL (Reinforcement Learning-based) end-to-end method [36], to infer the loop invariant of this example and show the weakness of the existing approaches. During the process, we give the intuitions and insights of our approach.

3.1 Problem Formulation

The verification of a program can be formalized with axiomatic semantics[15], which consists of a set of axioms and inference rules for proving the correctness of a program. Let P, Q, I denote predicates over program variables, S denote program statements, and B denote the loop condition. The Hoare triple $\{P\}\ S\ \{Q\}$ is valid if for any program state that make P true, after finishing executing S, can also make Q true. P is called *pre-condition*, and Q is called *post-condition*. When verifying a loop statement, we should utilize the rule below:

$$\frac{P \Rightarrow I \qquad \{I \wedge B\}\ S\ \{I\} \qquad (I \wedge \neg B) \Rightarrow Q}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{Q\}} \tag{1}$$

The loop invariant inference problem is: given P, Q, B, S, we need to find an I (namely, *loop invariant*) that satisfies the three conditions of the rule in Equation (1). This I is actually an abstraction of all possible program states at the loop location of the program. Once we find the I, we can apply (1) to verify the correctness of the loop statement.

Table 1: The motivation example.

```
// variable declarations
int x,y,n;
// pre-conditions
assume(n >= 0 && x == n && y == 0);
// loop body
while (x > 0) {
  x  = (x - 1);
  y  = (y + 1);
}
// post-condition
assert(y == n);
```

Table 1 gives an example which motivates the work in this paper. It is a simple program that only contains three variables and one loop. The variables are x, y, and n; the loop condition $B(x, y, n)$ is $(x > 0)$; the pre-condition $P(x, y, n)$ is $(x = n) \wedge (y = 0) \wedge (n >= 0)$; and the post-condition $Q(x, y, n)$ is $(y = n)$. We encode S as a predicate $S(x, y, n, x', y', n')$ where x', y', and n' represents the variable values after the update statement S is executed, i.e. $S(x, y, n, x', y', n')$ is $(x' = x - 1) \wedge (y' = y + 1) \wedge (n' = n)$. If the proposed $I(x, y, n)$ validates $P(x, y, n) \Rightarrow I(x, y, n)$, $I(x, y, n) \wedge B(x, y, n) \wedge S(x, y, n, x', y', n') \Rightarrow I(x', y', n')$, and $I(x, y, n) \wedge \neg B(x, y, n) \Rightarrow Q(x, y, n)$, then this $I(x, y, n)$ is an adequate loop invariant to verify the correctness of the example program.

The loop invariant of the above program is: $(x+y = n) \wedge (y \leq n)$. Although this invariant is very simple, it is not trivial to infer even in a linear hypothesis space, as we will see later.

3.2 ICE Template Iteration Method

The ICE Template Iteration Method [12] (ICE for short) works by iteratively proposing possible invariant templates and solving them according to the counterexamples. The hypothesis space of ICE is a series of invariant templates. An invariant template is a logical expression that consists of program variables, undecided constants, and logical functions and connectives. The series of limited linear templates ICE adopted is as follows:

$$\vee_i \wedge_j (s_1^{ij} * v_1^{ij} + s_2^{ij} * v_2^{ij} \leq c^{ij}) \tag{2}$$

v_k (k is 1 or 2) stands for one program variable, and s_k and c are constants waiting to be synthesized. This template is a disjunctive normal form, and each literal is a "less and equal"($\leq$) logic predicate

over a linear combination of two variables ($s_1^{ij} * v_1^{ij} + s_2^{ij} * v_2^{ij}$) and a constant ($c^{ij}$). The numbers of disjunctions and conjunctions are changeable. ICE template iteration always tries the simplest template (like $s_1 * x + s_2 * y < c$) first and then gradually tries more complex templates (templates with more disjunctions and conjunctions).

ICE template iteration uses an SMT solver like Z3 to solve the possible values of these unfixed constants s and c so that the solved candidate invariant must pass all previous counterexamples. A counterexample is a concrete value assignment for all relevant variables of the program. There are three kinds of counterexamples:

- positive counterexample (p): The counterexample falsifying $P \Rightarrow I$ is called *positive counterexample* (p) since p must make P true and make I false. In the next round, ICE needs to propose an invariant I which p makes it true.
- negative counterexample (n): The counterexample falsifying $(I \wedge \neg B) \Rightarrow Q$ is called *negative counterexample* (n), and ICE needs the next proposed invariant be false when assigned with n.
- inductive counterexample (i_1, i_2): The counterexample falsifying $\{I \wedge B\}\ S\ \{I\}$, called *inductive counterexample* (i), is a little different since it contains two assignments of the variables. The next invariant has to be false for the first assignment or be true for both assignments. The first assignment is denoted as i_1, and the second is denoted as i_2.

All counterexamples will be collected and maintained in three sets (CE_p, CE_n, and CE_i). Let V be a variable vector to represent the variables of the program. The next invariant I has to make the following logic expressions all valid.

$$\begin{aligned} &\forall p \in CE_p \quad V = p \Rightarrow I(V) \\ &\forall n \in CE_n \quad V = n \Rightarrow \neg I(V) \\ &\forall i \in CE_i \quad (V_1 = i_1 \wedge V_2 = i_2) \Rightarrow (I(V_1) \Rightarrow I(V_2)) \end{aligned} \tag{3}$$

Obviously, finding an invariant I for Equation (3) is easier than finding the I for the three conditions in Equation (1), since the three counterexample sets (CE_p, CE_n, and CE_i) are actually a "partial view" of the three verification conditions.

For our motivation example in Table 1, one of the simplest templates in ICE would be: $s_1 * x + s_2 * y \leq c$. Constants s_1, s_2, and c will be solved by an SMT solver so that the final invariant must be consistent with all counterexamples (in other words, making the three expressions in equation (3) all valid). In order to make the constraint solving process quicker, ICE also restricts the constants to a narrow range. For example, s_k is usually restricted to $[-1, 1]$.

Supposing we already have the counterexamples sets as:

$CE_p : \{x = 100, y = 0, n = 100\}$

$CE_n : \{x = 0, y = 1, n = 2;\ x = 0, y = 2, n = 1\}$

$CE_i : \{(x = 1, y = 0, n = 100;\ x = 0, y = 1, n = 100)\}$

Then, the SMT query used to infer the constants will be:

$assert((s_1 * 100 + s_2 * 0 \leq c) \wedge$

$\neg(s_1 * 0 + s_2 * 1 \leq c) \wedge$

$\neg(s_1 * 0 + s_2 * 2 \leq c) \wedge$

$((s_1 * 1 + s_2 * 0 \leq c) \Rightarrow (s_1 * 0 + s_2 * 1 \leq c))\ )$

Solving this constraint can get the values of s_1, s_2, and c as $-1, -1, -3$ that make the invariant consistent with all previous counterexamples. It should be noted that the quantifiers are eliminated in Equation (3) by substitution and expansion. Then, the next invariant candidate would be $(-1) * x + (-1) * y \leq -3$, and a new positive counterexample like $x = 2, y = 0, n = 2$ will be given by the SMT solver. The above process will then repeat again. After many iterations, the counterexamples will increase to an edge where the template $s_1 * x + s_2 * y \leq c$ can no longer be solved (unsatisfiable). When at that edge, the ICE template iteration method will try a more complex template such as $(s_3 * x \leq c_2) \vee (s_1 * x + s_2 * y \leq c_1)$.

It is noticed that the new invariant overcomes at least one more counterexample that is not overcome by previous invariants. Therefore, if the correct loop invariant can fit in the template, ICE Template Iteration has an intelligent strategy that makes candidate invariants towards the loop invariant step by step along with the growing volumes of the counterexample sets. Unfortunately, the ICE template iteration method failed to solve the motivation example in the time limit of 10 minutes (with Z3 as the solver), during which it tried out 10 different templates from simple to complex with 441 times of SMT queries. The suitable template to solve for the motivation example is at the very behind of ICE's attempting order.

Looking into its search strategy, we may notice that it treats the program and verification conditions as a black box, and the attempting orders of templates for all different programs are the same. Specifically, the ICE method exhaustively tries the templates from simple to complex. It will not try more complex templates until all simpler templates are proven unsatisfiable, while sometimes this un-satisfaction can only be reached after thousands of interactions between template solving and candidate checking. For the motivation example, suppose we have a template $x < c$. All we have to do is solving the constant c. Suppose we have one positive counterexample $x = 1, n = 1, y = 0$, then the template solving may give $x < 2$. The candidate checking then returns another positive counterexample $x = 2, n = 2, y = 0$. The template solving continues proposing $x < 3$. However, the fact is that x can be any value bigger than 0 as long as it is equal to n. Hence, this process will not stop until one counterexample reaches the maximum of integer, and the template is then proven unsatisfiable. This exhausting strategy may waste much time.

It is highly expected to have a more efficient strategy to adjust the templates to make the space search focused and quickly. It requires to utilize information not only from the counterexamples but also from features of the verification conditions and the program under verification.

3.3 End-to-End RL Inference

Reinforcement learning is adopted to infer loop invariant in [36] as an end-to-end framework (namely Code2Inv). As a *guess and check* approach, it generates candidate invariants according to BNF (Backus-Naur form) expansion rules selected by a deep RL model and checks the invariants against the verification conditions by SMT solving. Unlike the ICE method, which tries the templates inflexibly, the end-to-end deep RL model tries to learn heuristics from the program's source under verification by the reward feedback

from the SMT solver, which enables it to have customized search preferences for different programs.

End-to-end RL Inference treats an invariant as a fully expanded grammar tree. A limited linear grammar of invariant used by them is listed as the following BNF:

$$\begin{array}{lll} S & := & C|C \wedge C|C \wedge C \wedge C \\ C & := & (D)|(D \vee D)|(D \vee D \vee D) \\ D & := & exp\ cp\ s \\ exp & := & s * v|s * v + exp \\ cp & := & < | > | \leq | \geq | = \end{array}$$

The symbol s is a constant, and the symbol v is one variable from the program. The generation of candidate invariants is performed as a Markov Chain decision process. Begin with the start symbol S, the action of the RL model is to select an expanding rule within the rules whose lefthand is S. Then, the RL model repeatedly selects the rule for the leftmost non-terminal symbol. When there is no more non-terminal symbol, the candidate invariant is synthesized.

For example, they synthesized a candidate invariant for the motivation example in Table 1 as $(3 * x + 4 * y > 2)$. The full grammar expansion trace (also the action trace) of it is:

$S \rightarrow C \rightarrow (D) \rightarrow (exp\ cp\ s) \rightarrow (s * v + exp\ cp\ s)$
$\rightarrow (3 * v + exp\ cp\ s) \rightarrow (3 * x + exp\ cp\ s) \rightarrow (3 * x + s * v\ cp\ s)$
$\rightarrow (3 * x + 4 * v\ cp\ s) \rightarrow (3 * x + 4 * y\ cp\ s) \rightarrow (3 * x + 4 * y\ >\ s)$
$\rightarrow (3 * x + 4 * y\ >\ 2)$

The RL model took 11 sequential actions to synthesize $(3*x+4*y > 2)$. After a loop invariant candidate I is synthesized, the SMT solver will be used to check if this I makes all the three conditions in Equation (1) valid. If not, the RL model will receive a negative reward. Otherwise, the RL model will receive a positive reward. However, this reward mechanism is too sparse. Since most of the time, the RL model is in the process of "trial and error" and seldom receives positive rewards.

Therefore, a fine-grained reward mechanism is designed in [36] by utilizing the population information of the counterexample sets in equation (3), such as the percentage of passed counterexamples. In addition, heuristic rewards are also adopted. For example, they will punish the RL model if it synthesizes a contradiction or an invariant containing repeated literals.

With the rewards for the actions given, the learning parameters of the RL model will be updated by gradient descent. The updating policy used by them is called the advantage actor-critic (A2C) method, which is further discussed in Section 4.2.

Return to the motivation example, after the candidate invariant $(3 * x + 4 * y > 2)$ failed on the counterexamples and a negative reward was given, the parameters were updated. Then, in the next iteration, the RL model tried a different candidate as $((1*x+1*y = 5) \vee (1 * x + 1 * y < 0)) \wedge (1 * x + 1 * y = 1)$, which contained a contradiction and again did not pass the counterexamples, resulting a even less negative reward for punishing the contradiction. It took a long term of "trial and error" for the RL model to produce a candidate just to pass all previous counterexamples.

In fact, the action trace to generate the correct answer $(x + y == n) \wedge (y \leq n)$ is very long: 19 actions, which in the defined grammar form is: $(1*x+1*y+(-1)*n == 0) \wedge (1*y+(-1)*n \leq 0)$. If the RL model has not learned any heuristics yet (choosing grammar rules with equal probabilities), the expected probability of generating this correct invariant by chance is $\frac{1}{47239200}$, which is almost impossible. The RL model often needs to try out most of the wrong options to raise this probability gradually. As a result, Code2Inv failed to infer the correct loop invariant for the motivation example in a limited time (10 minutes).

Although the RL model does not need to try the candidates following a fixed order like the ICE method, it needs a long term of "trial and error" to find the direction even only to pass previous counterexamples. Hence, it occurs to us what if we use an RL model to generate templates and use an SMT solver to solve these templates. By doing so, we can shorten the action trace of the RL model, making it focus on cutting the wide space, and leave undecided factors (the small space) to SMT solving that can fully utilize the counterexamples. Taking the motivation example, suppose we already have enough counterexamples, the RL only need to produce a template as $(s_1 * x + s_2 * y + s_3 * n == s_4) \wedge (s_5 * y + s_6 * n \leq s_7)$ which can be generated with only 12 actions. After this template is solved, we will get the correct answer. The expected probability of producing such a template with no heuristics (equal randomly) is $\frac{1}{21600}$, 2187 times higher than the original end-to-end approach. In the experiments (see Section 5), our approach only take 11.31 seconds to solve this instance.

4 OUR APPROACH

The overall framework of our approach is depicted in Figure 1. First, we set up a general template (Section 4.1). Then, we use an RL model to prune the general template according to the program's features, verification conditions, and counterexamples (Section 4.2). After the general template is pruned to a sub-template, we use the SMT solver to solve this sub-template into a candidate and carry out candidate checking (Section 4.3). According to the outcomes, we provide the two-dimensional reward, and parameters are updated through gradient descent (Section 4.4). We expect to find the correct loop invariant by iteratively running the above process.

4.1 General Template and Sub-templates

We consider covering the QF-NIA[2] space. The calculation symbols (functions) we consider are $\{+, -, *, div, mod\}$. The comparison symbols (predicates) considered are $\{<, \leq, =\}$. The logical connectives are $\{\vee, \wedge\}$. We use v to represent a variable of the program under verification and s to represent a concrete constant or an undecided constant. The general template can be defined as:

$$\begin{array}{lll} Template & := & \bigwedge_{i=0}^{i=nc}(\bigvee_{j=0}^{j=nd_i} p_{ij}) \\ p & := & t < t|t \leq t|t = t \\ t & := & v|s|(t\ op\ t) \\ op & := & +|-|*|div|mod \end{array}$$

The general template is able to express almost every practical invariant in the QF-NIA theory. For practice, we set a maximum number of conjunctions and disjunctions (nc and nd) as 100. s can be either left along or decided with a concrete integer value between -10000 and 10000 or integers that appear in the program's source.

A ***sub-template*** refers to a shape-decided template where only the symbols of s (constants) are not decided. For example, $((((x * s_1)\ div\ y) + s_2) < (z\ mod\ 3))$ is a *sub-template* where x, y, and z are

the program variables, s_1 and s_2 are undecided constants, and 3 is a concrete constant.

4.2 Pruning

We design a reinforcement learning model to prune the general template. It prunes the template by iteratively selecting expansion rules for the left-most non-terminal symbols. The whole design is depicted in Figure 2. It contains five neural network modules:

E is the module that inputs the three counterexample sets: CE_p,CE_n, and CE_i, and outputs a feature vector of the counterexamples. We use PCA (Principal Component Analysis) to extract the most important counterexamples (as the volume of counterexamples will be very large after hundreds of iterations) and use a convolution module to convert them into a single feature vector.

G is the module that inputs the program and the feature vector output by E and outputs the feature vector of the program. We use two different representations of the program's source code. First, we transform the source code into static single assignment (SSA) [9] form and then constructs a control flow graph (CFG) on it (which is the same as [36]). Meanwhile, we also construct the SMT-LIB[30] format of the pre-conditions, post-conditions, loop condition, and transition from the source code. We use Tree-LSTM [37] to embed the logic expressions and use the attention mechanism to aggregate them into a single vector. In the end, we embed the three feature vectors into an overall feature vector ($feature_{vec}$) with the attention mechanism.

T is the module that inputs a template and outputs a vector representing the semantics of the template. We implemented T as a Tree-LSTM embedding the template (a partial logic expression) during pruning.

P is the critic module that inputs the state vector and outputs the predicted reward of this state. It is a two-layer, fully connected network that inputs the state vector and outputs the predicted reward.

π is the actor module that inputs the vectors output by G and T and outputs a probability distribution over available actions. π adopts an attention mechanism (weighted sum) to integrate the overall feature vector and the state vector into a "Action Vector" (A_{vec}) and uses Softmax to produce the "Action Distribution" (A_{dist}) over available actions. For example, if the leftmost non-terminal symbol is op, the available actions are $\{+, -, *, div, mod\}$, and the RL model only needs to select between the five actions. After we get A_{dist}, we use "Sampling" to sample one action (A) from the action distribution (A_{dist}) output by π. For example, suppose the action distribution produced by π is $\{+ : 0.1, - : 0.9, * : 0, div : 0, mod : 0\}$, then we will have a chance of 90% to sample the action $-$, 10% to sample +, and will never sample the other three. The decision-making process in π can be written as:

$A_{vec_i} := Sum_W(T(template_i), feature_{vec})$
$A_{dist_i} := Softmax(A_{vec_i} \times {rules_{vecs}}^T)$
$A_i := Sampling(A_{dist_i})$

where Sum_W stands for "weighted sum", $template_i$ is the current template produced by the previous actions, and $rules_{vecs}$ represents the feature vectors of available rules, whose shapes are (nor, dim) (the number of available rules and the dimension of the action vector). *Sampling* is to sample one action over a probabilistic distribution of actions. The selected action A_i is then used to update the current template (replace the leftmost non-terminal grammar node in the current template with the selected node). The above process will repeat till the template is pruned into a *sub-template*. The pruning process is described in Algorithm 1.

Algorithm 1 The process of pruning the general template (RL.prune)

Input: the general template $\mathcal{GT}$, program's source code S, counterexample sets CEs
Output: pruned sub-template $\mathcal{PT}$
/*begin with the general template*/
$cur\_T := \mathcal{GT}$.
repeat
  /*get the leftmost non-terminal symbol*/
  $handler$:= Get-LeftMost-nonsymbol(cur_T)
  /*list the available actions*/
  $availables$:= Available-Action-Selection($handler$)
  /*calculate the probability distribution over the actions*/
  A_{dist} := Prune(S, CEs, $availables$)
  /*sampling the action*/
  A_i := Sampling(A_{dist}, $availables$)
  /*prune the template according to the selected action*/
  cur_T := Expand(cur_T, $handler$, A_i)
until IsSubTemplate(cur_T) is $True$
$\mathcal{PT} := cur\_T$.

All the neural network modules are connected to each other. Their parameters will be updated by the gradient back-propagated from the same loss target. The "Instructive Distribution" is designed for the two-dimensional reward (see Section 4.4).

4.3 Template Solving and Candidate Checking

After we obtained a sub-template where only some constants s are left to be solved, following the ICE method [12], we use the counterexample sets (CE_p, CE_n, and CE_i) to infer the rest constants. We denote the *sub-template* as $\mathcal{PT}$, and the substitution operation as $Sub(X, Y)$, meaning substitute all variables in X with the assignment Y. For example, $Sub((s_1 * x + s_2 * y < s_3), \{x = 3, y = 4\}) = (s_1 * 3 + s_2 * 4 < s_3)$. Therefore, to overcome all counterexamples (i.e., satisfying the conditions in Equation (3)), we call the SMT solver to find a solution for the following query:

$$\begin{aligned} assert(& \bigwedge_{p \in CE_p} Sub(\mathcal{PT}, p) \wedge \\ & \bigwedge_{n \in CE_n} \neg Sub(\mathcal{PT}, n) \wedge \\ & \bigwedge_{i \in CE_i} Sub(\mathcal{PT}, i_1) \Rightarrow Sub(\mathcal{PT}, i_2)) \end{aligned} \tag{4}$$

If we get an "sat" return, we can use the solution sol (an assignment for all constants) to get the inferred loop invariant candidate: $I = Sub(\mathcal{PT}, sol)$. This invariant I is then sent for the final check. If the template solving runs out of time or returns "unsat", this *sub-template* is problematic, and we need to "re-prune" the general template till we can solve the *sub-template* successfully.

After the sub-template is successfully solved, we check if the candidate invariant I can validate the verification conditions in

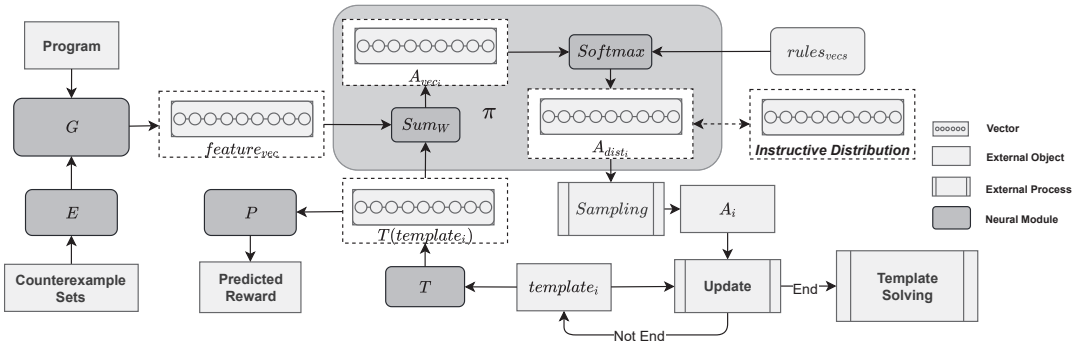

Figure 2: The architecture of the reinforcement learning policy network.

Equation (1). We use the following SMT query:

$$\begin{aligned} assert(&\neg(P(V) \Rightarrow I(V)) \\ &\vee \neg(I(V) \wedge B(V) \wedge S(V, V') \Rightarrow I(V')) \\ &\vee \neg(I(V) \wedge \neg B(V) \Rightarrow Q(V))) \end{aligned} \tag{5}$$

where V is the variable vector, P is the pre-condition predicate, Q is the post-condition predicate, B is the loop-condition predicate, and S represents the update statements in the loop, with V as the variables before the update statement is executed and V' as the variables after the update statement is executed. If this SMT query returns "sat", we get a solution *sol* that is the counterexample denying the correctness of candidate invariant I. This counterexample will be added to one of the counterexample sets (CE_p, CE_n, or CE_i) according to its kind. If this SMT query is proven as "unsat", which means the I validates the three verification conditions, and we find the correct invariant as I. The loop invariant inference succeeds.

4.4 Two-Dimensional Reward Mechanism

4.4.1 Case by Case Analysis. Unlike traditional reinforcement learning, where the reward is either "prize" or "punish", we design a mechanism that supports a 2-dimensional reward to reflect the information from SMT solving. Besides "prize"-or-"punish", we have a "stricter"-or-"looser" when we punish the RL model. Apart from telling the RL model that it made a wrong decision, we also indicate heuristics to avoid it in the next round. This heuristic instruction reward significantly alleviates the sparse reward problem discovered in Code2Inv as the SMT solver only gives "sat", "unsat" or "timeout" as a result and gives no reason why this result is given. We achieve this by analyzing outcomes from the SMT solver in both template solving and candidate checking case by case. There are 5 cases once the RL model finished pruning (see Figure 3).

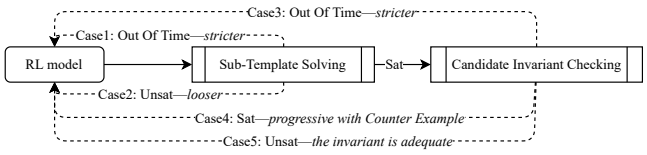

Figure 3: The five cases after a template is pruned.

The **first case** indicates that the SMT solver failed to solve the *sub-template* in the time limit (usually 10s). We can tell that the RL model must prune an over-complex *sub-template*. This is an early-failed case, so we punish it with a traditional negative reward: -10 (see Table 2). Also, we want to update the policy parameters of the RL model in a way that it can prune a simpler (on s) *sub-template* in the next round. That is, we prefer the RL model to set nc and nd_i with smaller values, expand t with fewer layers, choose more $+, -$ rather than $*, div, mod$, and leave less s undecided. We are sure that this outcome (OOT) is 100% due to over-complexity of the template. Therefore, we can punish the RL model and ask it to be "$stricter$" next time with high confidence "$very$". We achieve this by setting an "Instructive Distribution" (ID).

Table 2: The 1st-dimensional rewards for the five outcomes.

Cases	Description	1st Reward
C1	Template Solving is out of time.	-10
C2	Template Solving is unsatisfiable.	-10
C3	Candidate Checking is out of time.	-1
C4	Candidate Checking is satisfiable.	1
C5	Success.	10

IDs are manually set heuristics. We expect that the action distributions A_{dist} made by the RL model in the next round could be similar to the defined instructive distributions ID. The instructive distributions are listed in Table 3, where "N/A" represents no defined ID on that non-terminal, N is the maximum of conjunctions and disjunctions, which we set as 100 in this paper, and N_s is the number of possible concrete values for the symbol s. The numbers in Table 3 are quite easy to understand: they are probabilistic distributions over actions, where all numbers sum up to 1 in each tabular cell. When we want to raise the probability of selecting a specific action in the next iteration, we need to set a bigger number for that action in the ID (making other numbers in the distribution lower simultaneously). The actions that do not have a defined ID are actually not clearly related to the outcome. For example, no matter whether the template solving is out of time or unsatisfiable, we do not know how to change the actions on p ($t < t$, $t \leq t$, or $t = t$) to affect the result. Therefore, we give no instruction on these actions.

The **second case** shows that solving the template is proven unsatisfiable by the SMT solver. It is also an early-failed case. Moreover, it tells that we may cut too much space: no invariant that can pass all current counterexamples exists in the pruned search

Table 3: The Instructive Distributions (ID) (2nd-dimensional rewards) for the 3 negative cases.

	C1 (*stricter* on s, *very*)	C2 (*looser* on s, *medium*)	C3 (*stricter* on v, *little*)
$ID_{nc\|nd}$	$\{n : \frac{0.5^n}{1-0.5^N} \mid n = 1, 2, \ldots, N\}$	$\{n : \frac{2^{(n-1)}}{2^N-1} \mid n = 1, 2, \ldots, N\}$	$\{n : \frac{0.5^n}{1-0.5^N} \mid n = 1, 2, \ldots, N\}$
ID_p	N/A	N/A	N/A
ID_t	$\{v : 0.85,\ s : 0.1,\ (t\ op\ t) : 0.05\}$	$\{v : 0.1,\ s : 0.45,\ (t\ op\ t) : 0.45\}$	$\{v : 0.1,\ s : 0.85,\ (t\ op\ t) : 0.05\}$
ID_{op}	$\{+ : 0.4, - : 0.4, * : 0.1,$ $div : 0.05, mod : 0.05\}$	$\{+ : 0.05, - : 0.05, * : 0.4,$ $div : 0.25, mod : 0.25\}$	$\{+ : 0.4, - : 0.4, * : 0.1,$ $div : 0.05, mod : 0.05\}$
ID_s	$\{undecided : 0, concrete : \frac{1}{N_s}\}$	$\{undecided : 1, concrete : 0\}$	N/A

space. In other words, we want the RL model to cut less space. That is, the RL model should increase the value of nc and nd_i, expand t with more layers, try $*$, div, mod other than +, −, and leave more s undecided. However, this is not 100% the case. Another possibility is that we cut the space in the wrong direction, and the space size is not the reason. For example, suppose the correct *sub-template* is $t_1 \wedge t_2$, and we now have a *sub-template* like $t_1 \vee t_2 \vee t_3$. The template solver returns with "unsat". In that case, we should not loosen the *sub-template* but change the choices we made before. Hence, the reward we design for this outcome is: punish the RL model and ask it to be "*looser*" with confidence "*medium*", showing some hesitation in being looser.

The **third case** is a promising-failed case since the SMT solver finds a candidate that can pass all counterexamples. It also shows that checking the solved candidate is out of time, indicating that the loop invariant candidate must be too complex for the SMT solver to check the satisfaction. No matter whether the invariant candidate is the correct invariant or not, the only choice we have is to generate simpler candidates. That is, we want to generate simpler *sub-template*s. We should also notice that *sub-template* solving is successful, meaning that the *sub-template* is not over-complex for solving constants s. Therefore, replacing the appearance of v with more s is more likely the right direction. The appearances of v can also affect the complexity of the candidate invariant and are more likely to explain why *sub-template* solving succeeds but loop invariant checking fails. Hence, the reward of this outcome is designed as: punish the RL model and ask it to be "*stricter*" with confidence "*little*", focusing on reducing the appearance of v.

The **fourth case** indicates that we are progressing: although the loop candidate is denied with a counterexample, we are in fact a step closer to the answer by increasing the volume of counterexamples. Therefore, we should prize the RL model slightly, acknowledging its decisions. The **last case** shows that the candidate is adequate to verify the correctness of the program. All the actions made by the RL model can be recognized as "correct" actions. Therefore, The RL model should be prized greatly.

In general, we only need the numbers in IDs to reflect "how much looser or stricter you think the RL model should change according to the current outcome". With extremely sharp distributions, the RL model will change the pruning strategy rapidly and may take more iterations to find the sub-templates with proper complexity. With extremely blunt distributions, the RL model will change slowly even after facing the same failure many times. The sensitivity of learning can also be affected by learning rate and other hyperparameters in deep learning, which remains open research in the machine learning community. In this paper, we only give intuitive distributions reflecting our direct insights for the five outcomes.

4.4.2 Loss Design. Let the action sequence made by the RL pruner be $Seq_A = [A_1, A_2, ..., A_i, ..., A_n]$. First, the 1st-dim reward (noted as r, which is a number) will be back-propagated to each action with a discounter k, which we set to 0.95. The reward r_i for the ith action A_i is calculated as $r_i = k^{n-i} * r$. The reason is that although the full action sequence produced a bad (or good) sub-template, the actions are not the same bad (or good). This discounting is a popular heuristic in reinforcement learning [36].

Second, the 2nd-dim reward (ID_{A_i}) for the action A_i can be obtained by looking up the table (Table 3). For example, suppose the outcome is Case 1, the instructive distribution $ID_{nc|nd}$ for the action that acts on the non-terminal symbols nc or nd is: $ID_{nc|nd} = \{n : \frac{0.5^n}{1-0.5^N} \mid n = 1, 2, \ldots, N\}$. We use $symbol(A_i)$ to represent the non-terminal symbol on which A_i acts. We denote $ID_{A_i} = ID_{symbol(A_i)}$ for convenience.

After the rewards for each action are given, we can calculate the loss target. We mostly follow the advantage actor-critic (A2C) method[36], on which we aggregate the 2nd-dim reward. For each action, we have two rewards. The loss for a single action A_i for the 1st-dim reward r_i is written as:

$$\begin{aligned} Loss(A_i) =& MSE(r_i, P(T(template_i))) \\ &+ (r_i - P(T(template_i))) * C\_E(A_{dist_i}, F_{one}(A_i)) \end{aligned} \tag{6}$$

where r_i is the actual 1st-dim reward for action A_i. MSE stands for the Mean Square Error function, and C_E stands for the Cross-Entropy function. F_{one} inputs the A_i and outputs a one-zero vector where only the selected A_i is 1. For example, $F_{one}(op \rightarrow +) = \{+ : 1, - : 0, * : 0, div : 0, mod : 0\}$. P is the critic module, which predicates the expected reward. The intuition behind it is straightforward: first, we want the reward predictor as accurate as possible ($MSE(r_i, P(T(template_i)))$ as small as possible). Second, we want those actions that increase the rewards to be prized, which is to strengthen the policy network's mind (make $C\_E(A_{dist_i}, F_{one}(A_i))$ smaller).

Similarly, the loss for the 2nd-dim reward ID_{A_i} is written as:

$$Loss_{ID}(A_i) = \begin{cases} 0 & ID_{A_i} \text{is N/A} \\ C\_E(A_{dist_i}, ID_{A_i}) * \gamma & \text{else} \end{cases} \tag{7}$$

where γ is 0.1 for *very*, 0.05 for *medium*, and 0.01 for *little*. The overall loss can be defined based on Equation (6) and Equation (7)

as follows:

$$Loss = \sum_{i=0}^{i=n} (Loss(A_i) + Loss_{ID}(A_i))/n \quad (8)$$

After $Loss$ is computed, we update the parameters by gradient descent.

The overall process of our method is described in Algorithm 2. When calling "RL.learning.prize" or "RL.learning.punish" functions, the above loss calculation will be executed, and parameters will be updated. An illustrative running example process of our method can be checked in the supplementary material.

Algorithm 2 The Framework of RL-Pruning and SMT Solving

```
Input: the three loop invariant conditions VC, program's source code S
Output: the adequate loop invariant I
GT := general-template() /*get the general template*/
solved := False
CEs := Empty
while solved is False do
  /*template pruning*/
  PT := RL.prune(GT, S, CEs) /*call Alg 1*/
  /*template solving*/
  can_I := SMT-solver.solve(PT, CEs)/*get the candidate invariant*/
  if can_I is OOT then
    RL.learning.punish(Stricter, s, very) /*update parameters in Case1*/
  else if can_I is None then
    RL.learning.punish(Looser, s, medium) /*update parameters in Case2*/
  else
    /*template solving is successful*/
    ce := SMT-solver.verify(can_I, VC)
    if ce is OOT then
      RL.learning.punish(Stricter, v, little) /*update parameters in Case3*/
    else if ce is not None then
      CEs.add(ce)
      RL.learning.prise(little) /*update parameters in Case4*/
    else
      /*we find the answer*/
      solved := True
      I := can_I
      RL.learning.prise(very) /*update parameters in Case5*/
    end if
  end if
end while
```

5 EVALUATION

We implement our method as the tool named LIPuS[3]. The SMT solver we use is Z3. To evaluate our approach, we have four research questions: **RQ1** Is our method more efficient than the state-of-the-art methods? **RQ2** Is the designed two-dimensional reward mechanism necessary to improve search efficiency? **RQ3** Is the template solving necessary to improve the search efficiency? **RQ4** Do the manually designed Instructive Distributions generalize?

Benchmarks In order to answer the research questions, we design two sets of experiments: efficiency comparison with previous methods in linear and non-linear loop invariant inference. We prepared 124 linear benchmarks proposed in [32, 36] and 36 non-linear benchmarks proposed in previous works [36, 41], 3 of which are designed by us. All benchmarks consist of loops expressed as C code, annotated pre-conditions & post-conditions, and corresponding SMT-lib2[30] files. Each loop can have nested if-then-else blocks (without nested loops). Programs in the benchmark may also have uninterpreted functions (emulating external function calls) in branches or loop termination conditions. For CVC5, we develop a tool that can transform the SMT2-LIB2 format to the SyGuS-IF format, which can be checked in the code repository.

The 124 linear benchmarks are collected from recent literature [10, 13] (they contributed 46 and 40 benchmarks, respectively) and the SyGuS competition[6] by the authors of Code2Inv[36]. The SyGuS competition consists of 74 benchmarks. The authors of Code2Inv manually converted benchmarks from SyGus competition to C programs and removed the overlapped ones, resulting in 130 linear benchmarks. Later, the authors of CLN [32] discovered 9 invalid linear benchmarks (the programs are not correct) in these linear benchmarks and removed them. In the end, the rest 124 linear benchmarks became their linear test set as well as ours.

The 36 non-linear benchmarks come from three works. 26 benchmarks are from NLA[26], a benchmark of common numerical algorithms with non-linear invariants. The NLA contains 28 benchmarks where we removed 2 benchmarks that contain *double-type* variables and manually transformed nested loops into non-nested loops by introducing flag variables (taking only several days' work). We manually annotated post-conditions for NLA benchmarks according to the annotated loop invariants. 7 benchmarks are from Code2Inv [36], and 3 benchmarks are from us. The three non-linear benchmarks designed by us are relatively harder than the previous benchmarks, including the Cauchy-Schwarz inequality. As a result, our method solved one of the three benchmarks, and other state-of-the-art methods solved none.

Competitors For experiment competitors, we have selected 11 state-of-the-art methods. The competitors are listed in Table 4. A brief introduction of these methods can be checked in Section 2. We use their original experiment settings (i.e., hyperparameters and configurations) with only one exception: we extend the original linear templates of the ICE template iteration method with non-linear templates because it does not consider non-linear loop invariant inference in its original setting.

Evaluation Metrics To assess the efficiency of a loop invariant inference method, we choose to use the total inference time. This total inference time includes all costs taken to search the loop invariant practically: the time used in generating an initial example set, the time used in transforming the source code to other formats such as CFG, the time used in template solving and candidate checking, and any other time that is related to solve a specific program. The principle is to consider the time the method will take when it works in a real environment. If a method needs pre-training or pre-configuration, the time used in this preparing process should be ignored.

[3] https://doi.org/10.5281/zenodo.7909725

[6] http://sygus.seas.upenn.edu/SyGuS-COMP2017.html.

Table 4: The state-of-the-art methods for comparison.

Method	Description
Code2Inv [36]	an end-to-end RL method
CLN [32]	a deep learning method
GCLN [41]	a deep learning method
ICE-DT[13]	an ICE framework based decision tree method
ICE-DT-interval[40]	an ICE framework based decision tree method
ICE[12]	the original ICE framework method
SymInfer[27]	symbolic program trace based inferring.
Eldarica[16]	a model checking based Horn solver.
GSpacer[20]	a model checking based Horn solver.
CVC5[3]	a syntax-guided invariant synthesizer.
ImplCheck[31]	a specified solver for multi-phase CHC.

We measure this time by running different methods on the same computer and on the same benchmark sets. The quicker one is recognized as the better one. We set a total time limit of 10 minutes for each program. If a method takes more than 10 minutes, we recognize it as failing to solve the invariant, considering the benchmarks are relatively simple with an average lines of code less than 20. To alleviate the effect of randomness, we run each method five times, and calculate the average time.

The computing environment we set up for the experiments is as follows:

- CPU: AMD Ryzen Threadripper 3970X 32-Core
- GPU: Nvidia GeForce RTX 3090
- Memory: 96 GB
- OS: Windows 10 professional with subsystem for Linux 2

Training We do not adopt any pre-training. Instead, the whole process of LIPuS is an online learning process. We directly use LIPuS to solve the benchmarks, and the parameters are only initialized once for the first problem. Then, the parameters will be kept through the whole linear or nonlinear benchmarks. We set the hyperparameters according to the computing environment: the Template Solving time limit is set to 10s, the Candidate Checking time limit is set to 100s, the max number of counterexamples for Template Solving is 100, and the total inference time limit is set to 600s. For the neural network, the dimension for feature vectors is set to 128, and the number of PCA is set to 50. The optimizer we select is *Adam*. The deep learning package we select is *pytorch*. The Instructive Distributions used in different benchmarks are the same as in Table 3.

5.1 Result Analysis

The experiment results on linear and non-linear benchmarks are summarized in Table 5, where "ICE-DT-bd", "ICE-DT-dig", and "ICE-DT-varE" are three versions of ICE-DT-Interval [40]. If the time cost on an instance is over 600 seconds, we mark it as a failure and use 600 seconds when we calculate the average time. Our method LIPuS has solved all 124 linear problems and 25 non-linear problems. Specifically in the nonlinear benchmark, LIPuS solved 8 more nonlinear problems than the second-best one: SymInfer. In total, LIPuS solved the most instances among the state-of-the-art methods (see Figure 4, the total part).

Table 5: The summary of all competitor methods on linear and non-linear benchmarks.

Method	Linear#	Avg(s)	NL#	Avg(s)
ICE	58	320.88	0	timeout
ICE-DT	118	35.63	3	569.11
ICE-DT-bd	119	25.15	8	496.31
ICE-DT-dig	119	26.02	2	574.22
ICE-DT-varE	120	23.48	7	517.27
Code2Inv	97	170.77	3	579.70
CLN	6	571.08	0	timeout
GCLN[5]	39	412.05	0	timeout
SymInfer	36	430.37	17	336.63
Eldarica	**124**	3.57	8	472.55
CVC5	112	76.96	0	timeout
GSpacer	122	9.77	2	566.67
ImplCheck	**124**	**2.71**	10	434.40
LIPuS	**124**	23.49	**25**	**252.94**

5.1.1 Comparison On Linear Benchmarks. The accumulated time cost lines of all methods on linear benchmarks are depicted in Figure 4 (the linear part). The accumulated time cost line is made by accumulating the time cost of the competitor methods on each problem, and each time we add the least time cost. As can be seen in the picture, although our method spent a little more time on each problem, it can consistently solve every problem. On the contrary, methods like CLN, GCLN, and SymInfer are very quick to solve a small part of the problems but fail on the majority. In the setting of GCLN and SymInfer, they do not consider the post-condition of loop invariant. Instead, they aim to infer inductive invariants and compare them with the annotated known invariants for evaluation. However, in our setting, we need to prove the correctness of programs. We manually prepared the post-conditions for the benchmarks from NLA. The Z3 verifier will check the candidates with the three loop-invariant-conditions. Also, GCLN used its manually-defined "static-templates", which could be over-fitted to the benchmarks when solving invariants (see supplementary material). In our experiments, to make the comparison fair, we did not allow "static-templates". Otherwise, all tools can get the same good results with these "static-templates". ICE-DT-based methods are very efficient at solving these linear problems and solved only four problems fewer than ours. Compared to LIPuS, Eldarica and ImplCheck can also solve all instances but with much less time. The reason is that they have very good heuristics for linear problems such as Quantifier Elimination[5] which is no longer capable of nonlinear instances.

5.1.2 Comparison On Nonlinear Benchmarks. The accumulated time cost lines of all methods on the nonlinear benchmark are depicted in Figure 4 (the nonlinear part). Eldarica and ImplCheck, in spite of its outstanding performance on linear benchmarks, has only solved 8 and 10 nonlinear benchmarks. SymInfer performs the second-best on nonlinear instances with 17 solved (it proposed 31 candidates for the 36 nonlinear cases, but only 17 passed the

[5]It should be noted that GCLN is reported to solve 26 nonlinear benchmarks from 28 NLA benchmarks[26]. However, its experiment is carried out in a different setting. LIPuS can achieve the same performance in this setting. We put the details in the supplementary material.

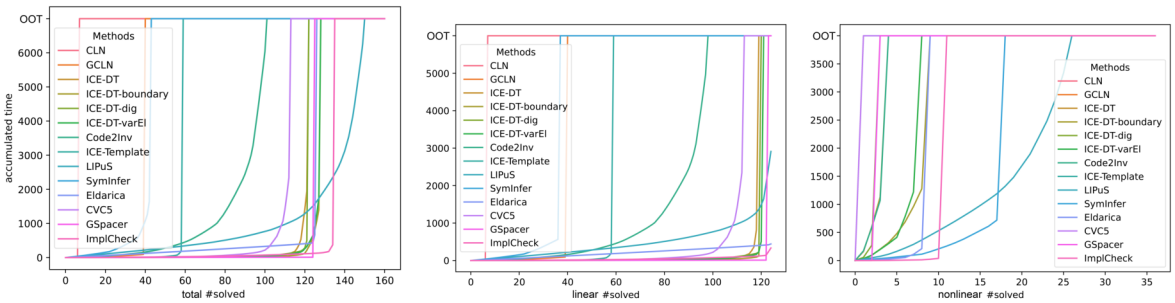

Figure 4: The accumulated time cost comparison on the linear and nonlinear benchmarks.

Table 6: The summary of all ablated LIPuS on linear and nonlinear benchmarks.

Method	Linear#	Avg (s)	NL#	Avg(s)
LIPuS-1-dim	111	85.29	7	529.80
LIPuS-noS	104	158.12	6	519.27
LIPuS-bald	94	181.44	1	589.49
LIPuS-full	**124**	**23.49**	**25**	**252.94**

verification). Compared to them, LIPuS has solved 25 nonlinear benchmarks. This line plot clearly illustrates our method's efficiency on nonlinear loop invariant inference. **Answer to RQ1:** Our method is not only more efficient on nonlinear loop invariant inference than the state-of-the-art methods but also performs well on the linear instances. Especially, we can solve 1.5x more instances than the second-best one, SymInfer, on nonlinear benchmarks.

5.2 Ablation Study

To evaluate the necessities of our designed modules and mechanisms, we carry out a series of ablation studies. First, we remove the two-dimensional reward mechanism and give the RL pruner only a one-dimensional reward to see the importance of this mechanism. That is, we set all $Loss_{ID}$ in Equation (7) to 0. The first ablated method is noted as "LIPuS-1-dim". Second, we want to investigate the necessity of template solving. For this purpose, we manually eliminate the RL pruning option of undecided constants, and as a result, all pruned templates will contain no constant waiting to be solved. The second ablated method is noted as "LIPuS-noS". Third, we remove both the 2nd reward and template solving, which results in an original end-to-end RL model. The third ablated method is noted as "LIPuS-bald". The original LIPuS is noted as "LIPuS-full".

We listed the results in Table 6. If we adopt a traditional one-dimensional reward, the RL pruner will spend a lot of time finding the right direction to control the complexity of pruned template, resulting in 13 linear and 18 nonlinear instances no longer solvable. If we ask the RL pruner to take the task that belongs to the SMT solving, the RL needs to decide every constant and can only solve 104 linear and 6 nonlinear benchmarks. Further, if we remove the two designs simultaneously, the RL model needs to search in a large space with extremely sparse rewards and fails more instances. **Answers to RQ2 & RQ3:** The two-reward mechanism is crucial to improving search efficiency. The template solving process is also essential. If these designs are removed from the framework, the performance will degrade significantly.

To further investigate the generalization ability of the proposed Instructive Distributions, we also carried out two additional experiments of stress testing against confounding variable inserting and CHC solving (see supplementary material). **Answer to RQ4:** The *ID*s used in the linear benchmark, the nonlinear benchmark, and the two additional experiments are exactly the same. Therefore, the experiment results show that the *ID*s designed by the case-by-case analysis generalize over different benchmarks.

6 THREATS TO VALIDITY

Three main threats threaten the validity of our work. First, the two-dimensional rewards are mostly based on manually defined heuristics. Different hyperparameter settings may lead to different experiment results. Although we evaluate the generalization ability of these hyperparameters, there is still a risk of over-fitting, as a universal problem existing in all deep learning methods. Second, the benchmarks may not be enough to thoroughly examine the loop invariant inference methods' performances. On the one hand, we need experts in program verification to annotate verification conditions to create benchmarks. On the other hand, the current loop invariant inference methods are rather primitive, and practical programs usually need to be filtered and modified as a benchmark. Third, our approach is highly dependent on SMT solvers in both template solving and candidate checking. However, recent studies [39] have discovered that SMT solvers may be problematic and return wrong answers in some cases. LIPuS has no power of resistance to these occasions.

7 CONCLUSION

By synergizing the RL-based pruning with SMT solving, we have extended the hypothesis space to the QF-NIA theory and kept the inferring time still acceptable. One essential reason for this success is the two-dimensional reward mechanism which enables the RL pruning model to produce *sub-templates* that are neither too strict nor over complex. However, our method is limited to integer-only programs and expects a small number of variables since our method is strongly dependent on the performance of SMT tools both in template solving and invariant candidate checking.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China under Grant No. 62032024.

REFERENCES

[1] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017. SyGuS-Comp 2017: Results and Analysis. In *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017 (EPTCS, Vol. 260)*, Dana Fisman and Swen Jacobs (Eds.). 97–115. https://doi.org/10.4204/EPTCS.260.9

[2] Thomas Ball and Sriram K. Rajamani. 2001. The SLAM Toolkit. In *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2102)*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.). Springer, 260–264. https://doi.org/10.1007/3-540-44585-4_25

[3] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[4] Cristiano Calcagno, Dino Distefano, and Viktor Vafeiadis. 2009. Bi-abductive Resource Invariant Synthesis. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5904)*, Zhenjiang Hu (Ed.). Springer, 259–274. https://doi.org/10.1007/978-3-642-10672-9_19

[5] B. F. Caviness and Jeremy R. Johnson. 1998. Quantifier Elimination and Cylindrical Algebraic Decomposition. In *Texts and Monographs in Symbolic Computation*.

[6] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50, 5 (2003), 752–794. https://doi.org/10.1145/876638.876643

[7] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen (Eds.). ACM Press, 269–282. https://doi.org/10.1145/567752.567778

[8] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 84–96. https://doi.org/10.1145/512760.512770

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991), 451–490. https://doi.org/10.1145/115372.115320

[10] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. 2013. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 443–456. https://doi.org/10.1145/2509136.2509511

[11] P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P. Madhusudan. 2018. Horn-ICE learning for synthesizing invariants and contracts. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 131:1–131:25. https://doi.org/10.1145/3276501

[12] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 69–87. https://doi.org/10.1007/978-3-319-08867-9_5

[13] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 499–512. https://doi.org/10.1145/2837614.2837664

[14] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2003. Software Verification with BLAST. In *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2648)*, Thomas Ball and Sriram K. Rajamani (Eds.). Springer, 235–239. https://doi.org/10.1007/3-540-44829-2_17

[15] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[16] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA Horn Solver. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, Nikolaj Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–7. https://doi.org/10.23919/FMCAD.2018.8603013

[17] Ranjit Jhala and Kenneth L. McMillan. 2006. A Practical and Complete Approach to Predicate Refinement. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3920)*, Holger Hermanns and Jens Palsberg (Eds.). Springer, 459–473. https://doi.org/10.1007/11691372_33

[18] Meir Kalech, Rui Abreu, and Mark Last. 2021. *Artificial Intelligence Methods for Software Engineering*. WorldScientific. https://doi.org/10.1142/12360

[19] Michael Karr. 1976. Affine Relationships Among Variables of a Program. *Acta Informatica* 6 (1976), 133–151. https://doi.org/10.1007/BF00268497

[20] Hari Govind Vediramana Krishnan, Yuting Chen, Sharon Shoham, and Arie Gurfinkel. 2020. Global Guidance for Local Generalization in Model Checking. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12225)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 101–125. https://doi.org/10.1007/978-3-030-53291-8_7

[21] Jiaying Li, Jun Sun, Li Li, Quang Loc Le, and Shang-Wei Lin. 2017. Automatic loop-invariant generation and refinement through selective sampling. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 782–792. https://doi.org/10.1109/ASE.2017.8115689

[22] Shang-Wei Lin, Jun Sun, Hao Xiao, Yang Liu, David Sanán, and Henri Hansen. 2017. FiB: squeezing loop invariants by interpolation between Forward/Backward predicate transformers. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 793–803. https://doi.org/10.1109/ASE.2017.8115690

[23] Kenneth L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2725)*, Warren A. Hunt Jr. and Fabio Somenzi (Eds.). Springer, 1–13. https://doi.org/10.1007/978-3-540-45069-6_1

[24] Kenneth L. McMillan. 2006. Lazy Abstraction with Interpolants. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4144)*, Thomas Ball and Robert B. Jones (Eds.). Springer, 123–136. https://doi.org/10.1007/11817963_14

[25] Kenneth L. McMillan. 2010. Lazy Annotation for Program Testing and Verification. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6174)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.). Springer, 104–118. https://doi.org/10.1007/978-3-642-14295-6_10

[26] 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, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 605–615. https://doi.org/10.1145/3106237.3106281

[27] ThanhVu Nguyen, Matthew B. Dwyer, and Willem Visser. 2017. SymInfer: inferring program invariants using symbolic states. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 804–814. https://doi.org/10.1109/ASE.2017.8115691

[28] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. *ACM Trans. Softw. Eng. Methodol.* 23, 4 (2014), 30:1–30:30. https://doi.org/10.1145/2556782

[29] Yihao Qin, Shangwen Wang, Kui Liu, Bo Lin, Hongjun Wu, Li Li, Xiaoguang Mao, and Tegawendé F. Bissyandé. 2022. Peeler: Learning to Effectively Predict Flakiness without Running Tests. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022*. IEEE, 257–268. https://doi.org/10.1109/ICSME55016.2022.00031

[30] Mukund Raghothaman and Abhishek Udupa. 2014. Language to Specify Syntax-Guided Synthesis Problems. *CoRR* abs/1405.5590 (2014). arXiv:1405.5590 http://arxiv.org/abs/1405.5590

[31] Daniel Riley and Grigory Fedyukovich. 2022. Multi-Phase Invariant Synthesis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 607–619. https://doi.org/10.1145/3540250.3549166

[32] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. 2020. CLN2INV: Learning Loop Invariants with Continuous Logic Networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=

HJlfuTEtvB

[33] Rahul Sharma and Alex Aiken. 2014. From Invariant Checking to Invariant Inference Using Randomized Search. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 88–105. https://doi.org/10.1007/978-3-319-08867-9_6

[34] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. 2013. Verification as Learning Geometric Concepts. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7935)*, Francesco Logozzo and Manuel Fähndrich (Eds.). Springer, 388–411. https://doi.org/10.1007/978-3-642-38856-9_21

[35] Rahul Sharma, Aditya V. Nori, and Alex Aiken. 2012. Interpolants as Classifiers. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 71–87. https://doi.org/10.1007/978-3-642-31424-7_11

[36] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 7762–7773. https://proceedings.neurips.cc/paper/2018/hash/65b1e92c585fd4c2159d5f33b5030ff2-Abstract.html

[37] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*. The Association for Computer Linguistics, 1556–1566. https://doi.org/10.3115/v1/p15-1150

[38] Shangwen Wang, Ming Wen, Bo Lin, and Xiaoguang Mao. 2021. Lightweight global and local contexts guided method name recommendation with prior knowledge. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 741–753. https://doi.org/10.1145/3468264.3468567

[39] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 718–730. https://doi.org/10.1145/3385412.3385985

[40] Rongchen Xu, Fei He, and Bow-Yaw Wang. 2020. Interval counterexamples for loop invariant learning. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 111–122. https://doi.org/10.1145/3368089.3409752

[41] 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 International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 106–120. https://doi.org/10.1145/3385412.3385986

[42] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 707–721. https://doi.org/10.1145/3192366.3192416

[43] He Zhu, Aditya V. Nori, and Suresh Jagannathan. 2015. Learning refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 400–411. https://doi.org/10.1145/2784731.2784766

Received 2023-02-16; accepted 2023-05-03