Inferring Loop Invariants for Program Verification: An Abductive Learning Perspective

Dai-Yang Luan [1] **Ming Li** [1]

Abstract

Loop invariant synthesis is one of the most important tasks in program verification. This problem is fundamentally undecidable, as it involves reasoning about infinite state spaces. Current software analysis techniques rely heavily on human-defined language biases to make the search tractable. When the language bias is misaligned with the unseen problem domain, these methods are prone to failure. Hence, to generalize on unseen problems, it is natural to explore training a model to learn and infer loop invariants. However, given the lack of supervision and the significant reasoning abilities required, learning to infer invariants is challenging. In this paper, we explore the feasibility of training a model for end-to-end loop invariant inference without human-labeled data. We present Abductive Loop InVariant lEarning (ALIVE) to combine machine learning and logical abduction in a mutually beneficial way. Logical abduction explores the solution space and subsequently refines the learning model by leveraging the discovered invariants. The machine learning model offers an initial solution that serves as a strong starting point, thereby facilitating the search process. Experiments show that ALIVE significantly outperforms other machine learning methods in both accuracy and efficiency.

1. Introduction

In order to improve the safety and reliability of modern software systems, an increasing number of researchers use formal methods to analyze whether the code given meets its requirements. For example, one may ask *does property*

```
void func(int n){
    int i = 0, a = 0, b = 0;
    assume(n >= 0);          // preconditions
    while (i < n){           // loop
        if (unknown()){
            a = a + 1;
            b = b + 2;
        } else{
            a = a + 2;
            b = b + 1;
        }
        i = i + 1;
    }
    assert(a + b == 3 * n); // postconditions
}
```

Figure 1. Program Verification. The goal is to prove the postcondition after executing the loop given that the precondition holds.

$\mathtt{P}(\mathtt{x},\mathtt{y})$ *hold after the execution of the loop*? As it is not known to us how many times the loop iterates, we should find a property that is maintained by the loop body (like a fixed point). Such a property is called a loop invariant. However, the problem of loop invariant generation is undecidable (Hoare, 1969), and it is in fact the hardest aspect of program verification (Garg et al., 2014) because given a loop invariant, the problem of verifying loop properties is reduced to checking the validity of specific verification conditions, which has been highly automated (Barnett et al., 2006).

The main challenge of loop invariant generation lies in the infinite state space of programs. Therefore, an important insight from software analysis literature is to find a finite number of abstract program states and reason about them (Hoare, 1969; Dijkstra, 1975). For example, it makes more sense to pay attention to *abstract states*, e.g., $\mathtt{x} \leq \mathtt{n}$, rather than *concrete states*, e.g., $\mathtt{x} = 1, \cdots, \mathtt{n}$. This is because abstract states represent higher-level properties or summaries of program behavior to avoid tracking every possible state the program could reach, which quickly becomes infeasible for large programs. Proper program abstractions can significantly reduce the scope of reasoning, narrowing down the infinite program state space to a manageable scale.

[1]National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China. Correspondence to: Ming Li <lim@nju.edu.cn>.

The second AI for MATH Workshop at the 42nd International Conference on Machine Learning, Vancouver, Canada.

Candidate Invariant I	$a+b=3i$	$(a+b=3i)\wedge(n\geq 0)$	$n>i$	$(a+b=3i)\wedge(n\geq i)$
Precondition Correctness	$\top$	$\top$	$(0,0,0,0)$	$\top$
Inductive Correctness	$\top$	$\top$	$(0,0,0,1)\rightarrow(1,2,1,1)$	$\top$
Postcondition Correctness	$(0,0,0,-1)$	$(0,3,1,0)$	$\top$	$\top$
Verification Result	False	False	False	True

Figure 2. Illustrations of candidates invariants and counterexamples. The candidate invariants correspond to the program in Figure 1. Counterexamples are written in terms of program states, i.e., assignments to variables (a, b, i, n). The symbol $\top$ illustrates that the verification condition is satisfied by the corresponding candidate invariant.

Many invariant inference methods generate program abstractions primarily based on human-crafted language bias, i.e., the hypothesis space from which potential solutions can be found. However, these approaches struggle to balance efficiency and generalizability. For example, Garg et al. (2014) solve invariants of the form $s_1v_1 + s_2v_2 \leq c, s_1, s_2 \in \{0, 1, -1\}$ quickly but fail if no proper answer exists in the hypothesis space while Sharma & Aiken (2016) use a more general template but their method converges slowly in complicated cases. Even the most advanced software analysis tools, such as ImplCheck (Riley & Fedyukovich, 2022), still rely on candidate program abstractions provided by annotators like Houdini (Flanagan & Leino, 2001) and are unable to search beyond the given candidate set. In order to alleviate the problem of the ultra-large hypothesis space brought by more general templates, Si et al. (2020) propose a reinforcement-learning-based method that improves efficiency by learning policies. However, it is essentially still a search-based method, where reinforcement learning only mitigates the time-consuming issue to some extent.

Given the success of learning models in code-related generation tasks like code summarization (Gao et al., 2023), it is natural to consider training an end-to-end model to generate invariants directly. However, different from code summarization, the task of invariant generation imposes stricter syntactical requirements and involves a generation process that demands reasoning ability rather than straightforward summarization. Moreover, obtaining ground truth labels for supervised learning is challenging due to the undecidability of the problem.

To address the issue, we consider learning to infer program invariants from an abductive learning perspective. Although formal tools cannot directly provide supervision information, they can determine the correctness of a candidate invariant and return counterexamples if the invariant is wrong, as shown in Figure 2. Based on this, logical abduction is proposed to refine the model-generated invariant based on the syntactical rules and with the help of the counterexamples. The revised invariant can then serve as a supervisory guide so that the model learns to infer loop invariants without the need for human-labeled ground truth data.

We propose Abductive Loop InVariant lEarning (ALIVE), a unified framework for loop invariant inference (Figure 3). ALIVE consists of two parts: a learning model and an abduction module. Each component plays an equally important role in the training stage. The abduction procedure, which does not support gradient descent, teaches the model by revising candidates generated by the model with symbolic manipulation. Meanwhile, the learning model, after preliminary training, provides the abduction module with candidates that contain more effective program properties, making it more likely to find suitable results. It is exciting to leverage machine learning and logic abduction in a mutually beneficial way.

Our contributions are summarized as follows:

- We introduce an abductive learning framework to address the problem of loop invariant inference with a formal definition provided.
- We present Abductive Loop InVariant lEarning (ALIVE) to combine inductive learning and abductive reasoning in a unified framework. We propose logical abduction to take advantage of the domain knowledge base, given that unlabeled data is readily available, but manual annotation is costly.
- Experimental results show that ALIVE significantly outperforms existing learning-based methods and is comparable with software analysis tools. Further more, its output is closer to answers of human experts than formal tools, even trained with no human-crafted features.

2. Problem Definition

We formally define the loop invariant inference task by introducing Hoare logic (Hoare, 1969). The fundamental idea behind Hoare logic is to express the correctness of a program in terms of logical assertions. An assertion is a logical statement about the program state that should be true at a particular point in the program. A Hoare triple, usually written as $\{P\}C\{Q\}$, is a formal way of expressing the relationship between logic assertions and code commands. The triple asserts that if the precondition P holds before

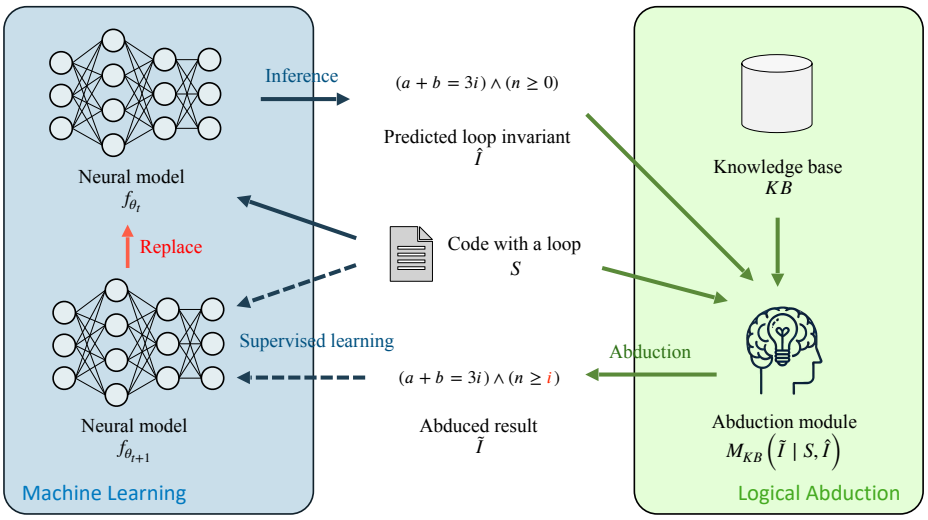

Figure 3. Illustration of our ALIVE framework. ALIVE consists of two parts: the neural model and the abduction module. The neural model generates initial candidate invariants, which are then refined by the abduction module using a knowledge base and counterexamples. The refined invariants are treated as pseudo labels to retrain the model, enabling a continuous improvement loop.

executing the command C, then the postcondition Q should hold after the execution of C.

Definition 2.1 (Loop invariants)**.** Suppose the Hoare triple of a loop is $\{P\}$ **while** (B) **do** $C\{Q\}$, there are three verification conditions (VCs) to be satisfied by a proper invariant I:

- Precondition correctness: $P \models I$.
- Inductive correctness: $\{I \wedge B\}C\{I\}$.
- Postcondition correctness: $I \wedge \neg B \models Q$.

A loop invariant typically describes a property that must be preserved by the loop as it iterates (precondition correctness and inductive correctness) and should be strong enough to entail the postcondition together with the exit condition of the loop (postcondition correctness). In Figure 1, regardless of which branch of the `if` statement is entered, the sum of a and b always increases by 3 while i increments by 1. Therefore, $(a+b=3i)$ is an important property preserved by the loop body. However, relying solely on this abstraction to form invariants is insufficient, as it may violate the postcondition correctness and result in counterexamples as shown in Figure 2. By adding $n \geq i$, the logical antecedent of the postcondition is $(a+b=3i)\wedge(n\geq i)\wedge\neg(i<n)$, which is equivalent to $(a+b=3i)\wedge(i=n)$ and is sufficient to entail the postcondition.

Loop invariants play a crucial role in program verification, and substantial effort has been invested in the task of loop invariant generation. However, these methods typically utilize domain-specific templates or heuristics and struggle in the tradeoff between efficiency and scalability. Given that the form of loop invariants is often closely related to program semantics, hopefully an end-to-end model can be utilized to learn such relationships.

Definition 2.2 (Learning to infer loop invariants)**.** Given a set of programs $\{S\}$ randomly sampled from some distribution $\mathcal{S}$, a model $f_\theta : \mathcal{S} \to \mathcal{I}$ is trained to map the program to its loop invariant with a knowledge base KB and the dataset $\{S\}$.

Unlike previous works, which leverage machine learning to optimize the search process for loop invariants, Definition 2.2 aims to train a model capable of directly generating invariants from source code. It is meaningful, as it does not rely on human-crafted search strategies or language bias. Instead, it demands fully leveraging past experience and knowledge base to train a model capable of generating invariants with sufficient proficiency. Due to the undecidability of the invariant inference problem, directly obtaining ground truth solutions as supervisory information is inherently difficult. In contrast, logical properties that loop invariants of a program S must satisfy are more readily obtainable with the knowledge base KB, denoted as $KB(S)$. We hereby formulate the problem to address as follows:

Definition 2.3 (Abductive perspective for learning to infer loop invariants)**.** Given $\{S\}, KB$ and an initial model f_{θ_0}, the problem is to maximize the consistency between the model prediction and the logical properties, i.e.,

$$\max_{\theta} \quad \mathbb{E}_S \left[\texttt{Con}\left(KB(S); f_\theta(S)\right)\right], \tag{1}$$

where `Con` is determined by a symbolic checker, its value

being 1 if $f_\theta(S)$ passes all verification conditions in $KB(S)$ and 0 otherwise.

3. Approach

We present ALIVE, an abductive-learning-based approach towards learning to infer loop invariants, as shown in Figure 3. In Definition 2.3, $KB(S)$ typically appear in a symbolic form, and their non-differentiable nature renders them unsuitable for direct use in training. To address this issue, we propose the abduction module in section 3.2 to indirectly incorporate the knowledge base into the model training process. In section 3.3, we describe the whole learning architecture of ALIVE.

3.1. Preliminaries

Invariant syntax. An invariant is a formula written in clausal normal form (CNF), i.e., the conjunction of one or more clauses, and a clause is the disjunction of one or more literals:

$$
\begin{aligned}
I &:= C_1 \wedge \cdots \wedge C_n \\
C &:= L_1 \vee \cdots \vee L_m \\
L &:= \mathit{Expr} \bowtie \mathit{Expr}, \bowtie \in \{\geq, >, =\} \\
\mathit{Expr} &:= \text{Variable} | \text{Constant} | \mathit{Expr} \oplus \mathit{Expr}, \oplus \in \{+, \times, \%\}
\end{aligned}
$$

Here we assume that all variables and constants other than 0 and 1 come from the code file.

Continuous consistency value with counterexamples. Candidate invariants with higher consistency should be prioritized. However, for incorrect invariants, the consistency in Definition 3.3 is always 0, even though some candidates are clearly closer to a correct solution. Motivated by (Sharma & Aiken, 2016), the consistency between candidate invariants and $KB(S)$ is measured based on counterexamples to obtain a continuous value, denoted as $\texttt{Con}(CE; I)$. Details about the calculation of $\texttt{Con}(CE; I)$ are presented in Appendix B.1, and its value satisfies the following three properties:

- $0 \leq \texttt{Con}(CE; I) < 1$ if $\texttt{Con}(KB(S), I) = 0$;
- $\texttt{Con}(CE; I) = 1$ if $\texttt{Con}(KB(S), I) = 1$;
- $\texttt{Con}(CE; I_1) > \texttt{Con}(CE; I_2)$ if I_1 is more likely to be closer to a correct solution based on the current set of counterexamples CE.

Knowledge base. The knowledge base KB comprises two aspects: the general definition of loop invariants (Definition 2.1) and the semantics of the programming language itself. In ALIVE, KB is implemented with tools for static analysis, including LLVM (Lattner & Adve, 2004) and Clang (Carruth et al., 2011). For a candidate invariant, all three verification conditions in $KB(S)$ are passed through an SMT solver. For each of the three verification conditions, the solver returns *syntax error* if there is a parse error, *logical invalidity* with a corresponding counterexample (as shown in Figure 2) or *satisfied*.

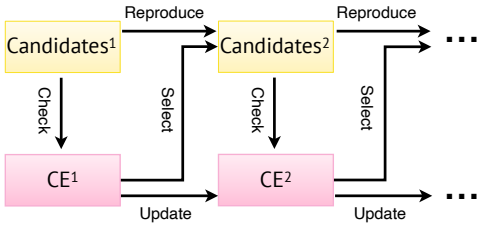

Figure 4. The abduction module workflow. It starts with the model-generated invariant and applies REPRODUCE for new candidates. Each candidate is checked against verification conditions, and a set of counterexamples, CE, is updated accordingly. CE is used to select candidates with higher consistency with the knowledge base KB. The process continues until a valid invariant is found or the iteration limit is reached.

3.2. Abduction with Knowledge Base

Given a program S and a model-predicted invariant $\hat{I} = f_\theta(S)$, the abduction module attempts to find a revised solution $\tilde{I}$ which is both syntactically correct and logically consistent with KB while keeping the highest similarity

$$
\max_{\tilde{I}} \quad \texttt{Con}\left(KB(S); \tilde{I}\right) \cdot \texttt{Sim}\left(\tilde{I}, f_\theta(S)\right). \tag{2}
$$

The similarity measure $\texttt{Sim}$ takes values between 0 and 1, and is inversely related to the number of revision steps. The objective is to identify a solution $\tilde{I}$ which satisfies $\texttt{Con}(KB(S); \tilde{I}) = 1$ within as few steps as possible.

To ensure syntactic correctness, $\hat{I}$ is replaced with an empty string if it is syntactically incorrect. Throughout the subsequent abduction process, syntactic correctness is maintained by all operations taken. For logical consistency, the abduction module (Figure 4) maintains a set of *candidates* and a set of counterexamples *CE*. The set of candidates is initialized with the model output $\hat{I}$, i.e., $\mathit{Candidates}^0 = \{\hat{I}\}$. REPRODUCE is performed on *candidates* to generate the next generation and halts once at least one correct solution is found in *candidate*. Like evolutionary algorithms, the REPRODUCE process involves the repetition of two operations: RECOMBINE and MUTATE.

RECOMBINE randomly selects two candidate invariants and, from the set of all clauses formed by both, randomly chooses elements to retain or delete. This results in a new candidate invariant. For example, if candidates $I_1 = A_1 \wedge A_2$ and $I_2 = B$ are selected, the output of RECOMBINE is randomly selected from the set $\{A_1, A_2, B, A_1 \wedge A_2, A_1 \wedge B, A_2 \wedge$

$B, A_1 \wedge A_2 \wedge B\}$.

However, relying solely on RECOMBINE cannot give rise to new clauses. Therefore, MUTATE is applied to the new candidate invariant with a certain probability. Types of mutations include:

- *Specialize* strengthens the candidate by deleting literals from clauses or adding a clause to the invariant.
- *Generalize* weakens the candidate by adding a literal to a clause or deleting a clause.
- *ExprShift* does not alter the number of literals or clauses. It replaces one expression with another.

REPRODUCE is repeated, with all newly generated invariants added to the population. A subset is selected to be retained, forming the next generation of the population. As discussed in section 3.1, the guidance provided by $\texttt{Con}\,(KB(S); I)$ is sparse and limited, and thus $\texttt{Con}(CE; I)$ is used as an alternative for selecting those with higher consistency. Candidate invariants in the new population are then checked to update the counterexamples. The above abduction process is repeated and terminates when a proper solution is found, or the maximum step is reached. Details about the algorithm can be found in Appendix B.2.

Note that multiple candidates may satisfy all verification conditions in the same population. In this case, the abduction module selects the shortest one. This is a simple bias, yet meaningful: by describing a loop invariant using fewer abstractions, the learning model is trained to focus on more essential properties of the program.

3.3. Model Training

In each iteration, the neural model f_θ is used to obtain candidate invariants $\{\hat{I}_i\}$, which are then revised by the abduction module to $\{\tilde{I}_i\}$. If a revised invariant satisfies all verification conditions, it is treated as a pseudo label and, together with its corresponding code file, added to the labeled training set $\mathcal{D}_t = \{(S_i, \tilde{I}_i)\}$, which is used for model training:

$$\theta_{t+1} = \arg\min_{\theta} \quad -\frac{1}{|\mathcal{D}_t|} \sum_{(S_i, \tilde{I}_i) \in \mathcal{D}_t} \log p_\theta \left(\tilde{I}_i | S_i \right). \quad (3)$$

In short, ALIVE works as follows: Given a program S, a neural model is used for obtaining a primitive candidate invariant $\hat{I}$, which is then taken as the input to the abduction module. With the knowledge base KB, the abduction module attempts to discover a correct solution $\tilde{I}$ in the neighborhood of $\hat{I}$. The abduced result is used for further model training.

4. Experiment

In this section, we verify the effectiveness of ALIVE in learning to infer invariants.

4.1. Experimental Setup

Dataset We evaluate ALIVE on 3,854 benchmark problems from SV-COMP [1]. The dataset is collected and preprocessed following Code2Inv (Si et al., 2020). In our dataset, a loop invariant inference problem is a C file composed of three parts: preconditions including variable assignments and assumptions, a loop body and postconditions in the form of assertions. Each C file corresponds to a file of SMT-LIB like format, which is logically equivalent.

Models In ALIVE, the learning model can be any fullability model capable of sequence generation. In this paper, we use CodeBERT (Feng et al., 2020) and CodeT5 (Wang et al., 2021). CodeBERT is an encoder-based pre-trained model on the transformer architecture. We extend it by adding 6 transformer layers as the decoder to perform invariant inference tasks. CodeT5 is another pre-trained model designed for code-related tasks.

Metrics We evaluate different generation methods by *solved percentage*. An instance is solved if the generation method finds or generates an invariant which satisfies all verification conditions within a certain time limit. The performance of learning models is measured by *test accuracy*. As there is no ground truth answer, the correctness of a generated invariant is checked by an SMT solver. To measure how similar two invariants are, we use the *Jaccard similarity*. It is calculated by the set of different literals the invariants contain

$$\text{Literals}(I) = \{L_k | \exists C_j \in I. L_k \in C_j\}. \quad (4)$$

Note that if $L_m \leftrightarrow L_n$, they are considered as the same literal written in different forms, and only one of them is added to the set. The Jaccard similarity between invariants I_1 and I_2 is

$$\text{Jaccard}(I_1, I_2) = \frac{|\text{Literals}(I_1) \cap \text{Literals}(I_2)|}{|\text{Literals}(I_1) \cup \text{Literals}(I_2)|} \quad (5)$$

4.2. Comparison with State-of-the-art

Loop invariant synthesis has long been a significant issue in the field of software analysis, attracting attention from researchers across various domains. A large number of methods have emerged, which can broadly be categorized into those based on random search, reinforcement learning, large language models, and static and dynamic

[1] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

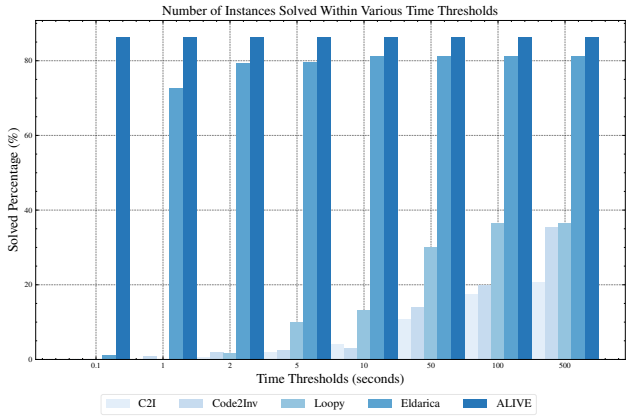

Figure 5. Comparison of ALIVE (CodeT5) with different kinds of state-of-the-art methods. ALIVE achieves the highest solved percentage (86.45%) while maintaining significantly lower computation time compared to search-based and reinforcement learning methods.

	Solved (%)	Avg Time (s)
C2I	20.81	82.70
Code2Inv	35.49	128.04
Loopy (GPT-4)	36.66	21.27
Eldarica	81.24	1.18
ImplCheck	81.29	5.45
ALIVE (CodeT5)	**86.45**	**0.09**

Table 1. Average time cost per solved instance with a time limit of 500 seconds. As an abductive-learning-based method, ALIVE performs better in terms of both solved percentage and efficiency.

analysis. We have selected five SOTA methods from these categories for comparison. C2I (Sharma & Aiken, 2016) is a random-search-based method using a Markov Chain Monte Carlo (MCMC) sampler to adjust the candidate. Code2inv (Si et al., 2020) solves the problem by multi-step decision making, where the policies are learned via reinforcement learning. Loopy (Kamath et al., 2023) uses large language models like GPT-4 (OpenAI, 2023) to generate invariants and adjusts the invariants by generating prompts based on verification results. Eldarica (Hojjat & Rümmer, 2018) is a model-checking-based Horn clauses solver and is commonly used for software analysis and verification. ImplCheck (Riley & Fedyukovich, 2022) is another software-analysis-based tool, which presents an SMT-based approach to synthesize implication invariants for multi-phase loops.

We first study the capability of different generation methods. As they employ diverse approaches and some methods take time to reach their full potential, solved percentage within different time limits is employed as a uniform metric. The time limit ranges from 0.1 to 500 seconds, which is acceptable in most cases. To make a fair comparison among the methods, we apply cross-validation over the whole dataset and results are shown in Figure 5 and Table 1.

The performance of ALIVE significantly outperforms other machine learning models and achieves comparable results with program analysis methods. Benefiting from the human-crafted language bias, Eldarica is capable of solving some of the instances efficiently when the required program abstractions happen to align with the predefined abstraction strategies. For other instances, in contrast, it takes a long time to synthesize a proper invariant. Machine learning-based methods, such as Code2Inv, aim to learn a more general strategy. However, the efficiency issue becomes more pronounced as they require continuous interaction with the verification tool to learn on a case-by-case basis. Similarly, large language models like GPT-4 struggle with this task due to the amount of time required for test-time revisions. Therefore, we believe that leveraging knowledge bases and unlabeled data to learn to infer loop invariants is promising. Training an end-to-end model alleviates efficiency issues, and leveraging domain knowledge bases reduces reliance on manual annotations or human-crafted language biases, making our framework more efficient and generalizable.

4.3. Ablation Study: Learning and Reasoning Benifit Each Other

Abduction helps the model learn better We replace the abduction module with a label generator to verify the effectiveness of logical abduction. Specifically, we treat software analysis tools as a *teacher* and train the learning models in a fully supervised manner. Eldarica and ImplCheck are run on the training set with a time limit of 10,000 seconds per sample to obtain efficient labels. 90.26% and 92.57% of the training set get labeled, and are used for training models respectively. Table 2 shows the test accuracy of the supervised approaches and ALIVE over different base models. Despite the absence of initial labels, the performance of ALIVE exceeds that of models trained with full supervision by up to 8.53%. Therefore, ALIVE implicitly enables the learner to explore the solution space more effectively than methods that explicitly constrain the learner to fit generated labels.

Model output enhances logical abduction Initially, an under-trained model can only generate random sequences, which provides limited utility for the abduction module. In this sense, almost all weak supervision information for the machine learning model comes from the abduction module, but the relationship between them goes beyond a teacher and student. To validate this, we obscure the output of the learning model from the abduction module and abduction is done with random initial candidates. As shown in Figure 6 (a)

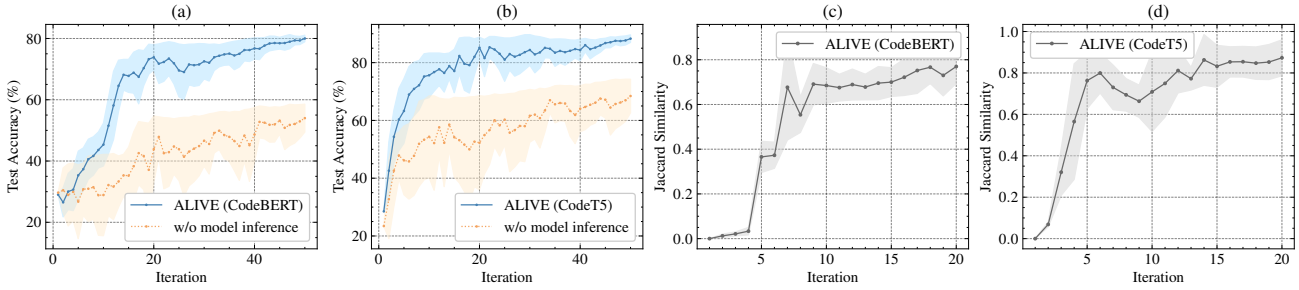

Figure 6. Illustrations of the learning mechanism in ALIVE. In (a) and (b), there is a significant degrade of learning performance when model predictions are not fed into the abduction module. In (c) and (d), the Jaccard similarity between model predictions and the abduced result is generally increasing, which indicates that fewer revisions are done by the abduction module while the model gets trained.

Base Model	Supervised by		ALIVE
	Eldarica	ImplCheck	
CodeBERT	75.78	78.90	**81.06**
CodeT5	77.92	83.07	**86.45**

Table 2. Comparison of test accuracy (%) with fully supervised approaches. ALIVE achieves higher test accuracy than models trained with supervision from software analysis tools (Eldarica and ImplCheck).

and (b), ALIVE achieves higher test accuracy than the case without model inference. This phenomenon is particularly significant when CodeBERT is used as the base model, and the final test accuracy difference between the two reaches 22.51%. For CodeT5, the decrease is 15.36%. Differences between the two curves are relatively small in the stage of cold start. However, as the abduction module without model inference depends heavily on hitting the answer in the neighborhood by chance, the supervision information it can provide to the learning model is very limited. This leads to a noticeable decrease in its performance upper bound in later iterations.

The role of learning models in the training stage To take a deeper look at how ALIVE works, we measure the Jaccard similarity of model inference and abduction result for instances over $\Delta\mathcal{D}_t$, a set of instances newly added to the labeled set

$$\Delta\mathcal{D}_t = \{(S_i, I_i) \in \mathcal{D}_t | \forall I \forall t' < t.(S_i, I) \notin \mathcal{D}_{t'}\} \quad (6)$$

It is calculated after each iteration and the result is shown in Figure 6 (c) and (d). In rounds of iterations, this similarity generally shows an upward trend. This indicates that the learning model, as a student supervised by the abduction module, is continuously improving its ability to infer loop invariants. Meanwhile, as the similarity increases, it is more likely that the correct answer falls within the neighborhood of the model inference. As a teacher, the learning model reduces the reliance of the abduction module on happening to hit a correct answer.

In this sense, the relationship between the learning model and the abduction module goes beyond a simple teacher-student dynamic. Their interaction is characterized by mutual benefit, where each component supports and enhances the other, leading to a more effective and adaptive training process.

4.4. Human Evaluation

Since there is no ground truth answer, any invariant that satisfies the verification conditions is considered correct. However, some of these invariants are excessively verbose, which increases the complexity of verification. Their lengthiness stems from numerous unnecessary assertions that do not effectively capture the relationship between the loop and its invariant. To assess the quality of invariants generated by different methods, we selected 100 instances from the test set and invited 5 human experts to annotate them. We compare the average Jaccard similarity of the human-annotated labels with those produced by state-of-the-art methods.

Table 3 presents the results of this human evaluation. Although Eldarica and ImplCheck demonstrate respectable accuracy, their solutions do not resemble those produced by human experts. They often sacrifice simplicity and efficiency to achieve greater generality. In Figure 8 of Appendix C, Eldarica has to traverse all concrete states to synthesize the final invariant and in Figure 10, the solution of ImplCheck is verbose, with many abstractions irrelevant to the verification of program properties. Invariants generated by ALIVE are more similar to human expert annotations. Even in cases where the results are incorrect, there is still a certain level of similarity, indicating that ALIVE is capable of finding suitable program abstractions.

	Solved	Unsolved	Average
C2I	0.3142	0.1434	0.2100
Code2Inv	0.2035	0.0923	0.1579
Eldarica	0.3891	-	-
ImplCheck	0.1631	-	-
Loopy (GPT-4)	0.5249	**0.4145**	0.4549
ALIVE (CodeT5)	**0.5926**	0.2072	**0.5324**

Table 3. Human evaluation with Jaccard similarity. The time limit for solving the instances is 10,000 seconds. We use the best answer found by C2I and Code2Inv to compare the similarity as unsolved instances and directly use the generated answers which fail to satisfy the verification conditions as unsolved instances of ALIVE.

Moreover, although the outputs of Loopy (GPT-4) exhibit high naturalness, they fail to provide correct solutions in most cases (Figure 10 and Figure 11). Even when formal tools are used to generate prompts for revision, the improvement remains minimal as there is no guarantee that previous mistakes are properly corrected. In contrast, models trained with ALIVE, by maximizing the consistency with the knowledge base through logical abduction, are more likely to generate correct solutions in an end-to-end manner.

5. Related Work

Loop invariant generation As an important problem in automated program verification, there have been many attempts to generate loop invariants. Most of existing loop invariant methods are search based, and the difficulty lies in the infinite target space of potential invariants. These approaches involve abstract-interpretation-based methods (Cousot & Cousot, 1977; Karr, 1976; Cousot & Halbwachs, 1978; Cousot & Cousot, 1979), interpolation-based methods (McMillan, 2003; Alberti et al., 2012), symbolic execution-based methods (Boyer et al., 1975; Nguyen et al., 2017), decision-tree-based methods (Garg et al., 2014; 2016), etc. These methods either introduce specific language biases to make the search feasible or manually design deterministic strategies for searching within an infinite space. To address the efficiency and generalization issue brought by deterministic strategies, Code2Inv (Si et al., 2020) is proposed to learn policies to generate invariants efficiently via reinforcement learning. Pei et al. (2023) uses large pre-trained language models for invariant generation. However, the capability of the learning model is constrained by Daikon (Ernst et al., 2007), the label generator for their method. In our method, an unlabeled dataset and an automated theorem prover like Z3 (De Moura & Bjørner, 2008) is sufficient to train a model for loop invariant generation.

Learning for code-related tasks The intersection of machine learning and software engineering has seen rapid advancements in recent years. Several learning models have been proposed that leverage machine learning to automate and enhance various code-related tasks such as program synthesis (Austin et al., 2021; Nijkamp et al., 2022), code translation (Lachaux et al., 2020; Zhu et al., 2022), bug localization (Liang et al., 2022) , and code summarization (Gao et al., 2023). Inspired by the success of transformer-based pre-trained models such as BERT (Devlin et al., 2019) and T5 (Raffel et al., 2020) in natural language processing tasks, many pre-trained models, including CodeBERT (Feng et al., 2020) and CodeT5 (Wang et al., 2021), have been developed using these architectures and have achieved outstanding results in many downstream tasks.

Neural-symbolic combination The combination of machine learning and reasoning is one of the key challenges in artificial intelligence today and efforts have been made to address it by various communities. Among them, learning-for-reasoning approaches take advantage of neural networks to assist in finding solutions (Evans & Grefenstette, 2018; Zhang et al., 2019), while reasoning-for-learning approaches utilize symbolic knowledge in the learning process to enhance the learning ability of neural systems (Xie et al., 2019; Xu et al., 2018).The above two types of approaches attempt to combine neural systems and symbolic systems by adapting one to the other, which fails to maximize the strengths of these two paradigms. Abductive learning (ABL) (Zhou, 2019; Dai & Zhou, 2017), as a learning-reasoning approach, balances the participation of neural systems and symbolic systems in the process of training and inference, and is thus beneficial for both sides. The ABL framework is renowned for its expressive and flexible nature, as it can be applied to both labeled and unlabeled data with an appropriate knowledge base. Dai & Muggleton (2021) enhance the ABL framework's ability to induce knowledge from the raw data, the optimization builds upon the EM algorithm.

6. Conclusion

We study the problem of inferring loop invariants for program verification from an abductive learning perspective. The proposed framework enables the model to learn to infer loop invariants by exploiting the knowledge base via logical abduction. The learning model trained by ALIVE is capable of solving a comparable number of instances as existing software analysis tools while significantly improving efficiency. It has also outperformed state-of-the-art learning-based methods. Experimental results suggest that the proposed ALIVE framework combines machine learning and logical abduction in a mutually beneficial way, taking advantage of domain knowledge and past experience. Compared with other methods, model predictions are more similar to the annotations of human experts, reflecting more essential properties of programs. Our work has demonstrated the

feasibility of learning-based invariant inference and paves the way for scalable, general and human-independent verification methods.

Impact Statement

Program verification is a crucial stage in software development. Loop invariant inference is among the most important tasks in program verification. In this paper, we introduce ALIVE, a unified framework for learning to infer loop invariants. We anticipate that this work will not introduce any negative ethical or social impacts.

References

Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., and Sharygina, N. Safari: Smt-based abstraction for arrays with interpolants. In *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings 24*, pp. 679–685. Springer, 2012.

Alur, R., Fisman, D., Singh, R., and Solar-Lezama, A. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438*, 2017.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. Boogie: A modular reusable verifier for object-oriented programs. In *4th International Symposium on Formal Methods for Components and Objects*, pp. 364–387. Springer, 2006.

Boyer, R. S., Elspas, B., and Levitt, K. N. Select—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10(6):234–245, 1975.

Carruth, C., Lattner, C., and Adve, V. Clang: A c language family front-end for llvm, 2011. http://clang.llvm.org.

Cousot, P. and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, 1977.

Cousot, P. and Cousot, R. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 269–282, 1979.

Cousot, P. and Halbwachs, N. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 84–96, 1978.

Dai, W.-Z. and Muggleton, S. Abductive knowledge induction from raw data. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, pp. 1845–1851, 8 2021.

Dai, W.-Z. and Zhou, Z.-H. Combining logical abduction and statistical induction: Discovering written primitives with human knowledge. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

De Moura, L. and Bjørner, N. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Minneapolis, MN, USA*, pp. 4171–4186. Association for Computational Linguistics, 2019.

Dijkstra, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8): 453–457, aug 1975.

Dillig, I., Dillig, T., Li, B., and McMillan, K. Inductive invariant generation via abductive inference. *Acm Sigplan Notices*, 48(10):443–456, 2013.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

Evans, R. and Grefenstette, E. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61:1–64, 2018.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. CodeBERT: A pre-trained model for programming and natural languages. In Cohn, T., He, Y., and Liu, Y. (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, November 2020.

Flanagan, C. and Leino, K. R. M. Houdini, an annotation assistant for esc/java. In *International Symposium of Formal Methods Europe*, pp. 500–517. Springer, 2001.

Gao, S., Gao, C., He, Y., Zeng, J., Nie, L., Xia, X., and Lyu, M. Code structure–guided transformer for source

code summarization. *ACM Transactions on Software Engineering and Methodology*, 32(1):1–32, 2023.

Garg, P., Löding, C., Madhusudan, P., and Neider, D. Ice: A robust framework for learning invariants. In *26th International Conference on Computer Aided Verification*, pp. 69–87. Springer, 2014.

Garg, P., Neider, D., Madhusudan, P., and Roth, D. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices*, 51(1):499–512, 2016.

Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

Hojjat, H. and Rümmer, P. The eldarica horn solver. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pp. 1–7. IEEE, 2018.

Kamath, A., Senthilnathan, A., Chakraborty, S., Deligiannis, P., Lahiri, S. K., Lal, A., Rastogi, A., Roy, S., and Sharma, R. Finding inductive loop invariants using large language models. *arXiv preprint arXiv:2311.07948*, 2023.

Karr, M. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

Lachaux, M.-A., Roziere, B., Chanussot, L., and Lample, G. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems*, 2020.

Lattner, C. and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, pp. 75, USA, 2004.

Liang, H., Hang, D., and Li, X. Modeling function-level interactions for file-level bug localization. *Empirical Software Engineering*, 27(7):186, 2022.

McMillan, K. L. Interpolation and sat-based model checking. In *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings 15*, pp. 1–13. Springer, 2003.

Nguyen, T., Dwyer, M. B., and Visser, W. 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*, pp. 804–814, 2017.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

Pei, K., Bieber, D., Shi, K., Sutton, C., and Yin, P. Can large language models reason about program invariants? In *International Conference on Machine Learning*, pp. 27496–27520. PMLR, 2023.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21 (140):1–67, 2020.

Riley, D. and Fedyukovich, G. Multi-phase invariant synthesis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 607–619, 2022.

Sharma, R. and Aiken, A. From invariant checking to invariant inference using randomized search. *Formal Methods in System Design*, 48:235–256, 2016.

Si, X., Naik, A., Dai, H., Naik, M., and Song, L. Code2inv: A deep learning framework for program verification. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, pp. 151–164, 2020.

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Xie, Y., Xu, Z., Kankanhalli, M. S., Meel, K. S., and Soh, H. Embedding symbolic knowledge into deep networks. *Advances in neural information processing systems*, 32, 2019.

Xu, J., Zhang, Z., Friedman, T., Liang, Y., and Broeck, G. A semantic loss function for deep learning with symbolic knowledge. In *International conference on machine learning*, pp. 5502–5511. PMLR, 2018.

Zhang, Y., Chen, X., Yang, Y., Ramamurthy, A., Li, B., Qi, Y., and Song, L. Efficient probabilistic logic reasoning with graph neural networks. In *International Conference on Learning Representations*, 2019.

Zhou, Z.-H. Abductive learning: towards bridging machine learning and logical reasoning. *Science China Information Sciences*, 62:1–3, 2019.

Zhu, M., Suresh, K., and Reddy, C. K. Multilingual code snippets training for program translation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 36, pp. 11783–11790, 2022.

Appendix

The structure of this appendix is as follows:

- Appendix A describes experimental settings;
- Appendix B introduces details about the counterexamples and the mutation mechanism;
- Appendix C performs case study over different loop invariant inference methods.

A. Experimental settings

A.1. Dataset

A well-known dataset is created by the authors of Code2Inv (Si et al., 2020), collected from related works (Garg et al., 2016; Dillig et al., 2013) and the 2017 SyGuS competition (Alur et al., 2017). The dataset consists of 124 valid instances. They are relatively simple and all of the instances have been solved by Eldarica (Hojjat & Rümmer, 2018) and ImplCheck (Riley & Fedyukovich, 2022). Therefore, we create a dataset with 284 instances from Competition on Software Verification, which is commonly used in loop invariant literature and publicly available. We follow the experimental settings and all benchmarks are pre-processed with the tool introduced in Code2Inv. Finally, we get 3,854 instances. The source of our dataset is shown in table 4.

Category	Count
bitvector	12
bitvector-loops	1
ldv-linux-3.4-simple	3
ldv-linux-3.7.3	2
ldv-linux-4.2-rc1	2
libvsync	2
loop-acceleration	4
loop-crafted	2
loop-industry-pattern	2
loop-invariants	9
loop-invgen	10
loop-lit	24
loop-new	9
loop-simple	2
loop-zilu	51
loops	12
loops-crafted-1	23
nla-digbench	17
nla-digbench-scaling	72
psyco	4
recursified_loop-crafted	2
recursified_loop-invariants	3
recursified_loop-simple	2
termination-nla	14

Table 4. Benchmark Categories and Counts from `sv-benchmarks-main/c/`.

A.2. Computing Infrastructure

All methods are run on a server with 4 AMD EPYC 7H12 64-Core CPUs and 8 NVIDIA A100-PCIE-40GB GPUs. The operation system is Linux version 5.15.0-105-generic. ALIVE is implemented in Python language and evaluated with Python 3.10.9 and PyTorch 2.3.0.

A.3. Cross Validation

A uniform experimental setting is needed for a fair comparison, as compared methods apply diverse performance optimizations. For example, while ALIVE benefits from offline training, Loopy requires well-trained LLMs, and tools like Eldarica benefits from human-crafted features and optimization strategies. In this paper, if a method needs pre-training or pre-configuration, the time used in this preparing process is ignored.

The original 284 examples are divided into 10 parts, each of which consists of 28 or 29 examples. In each validation iteration, 9 parts of them are used as the training set and 1 part is used as the test set. Note that each example is paired with all its augmented versions, i.e., if an example appears in the training (test) set, its augmented versions will also appear in the training (test) set.

B. Implementation Details

B.1. Counterexamples and Consistency Calculation

```
void func(int n){
    int i = 0, a = 0, b = 0;
    assume(n >= 0);          // preconditions
    while (i < n){           // loop
        if (unknown()){
            a = a + 1;
            b = b + 2;
        } else{
            a = a + 2;
            b = b + 1;
        }
        i = i + 1;
    }
    assert(a + b == 3 * n); // postconditions
}
```

A proper loop invariant I should satisfy:
Precondition Correctness (VC1):

$$(x = 0) \rightarrow I[x, n]$$

Inductive Correctness (VC2):

$$((x < n) \wedge I[x, n] \wedge (x' = x + 1)) \rightarrow I[x', n]$$

Postcondition Correctness (VC3):

$$(I[x, n] \wedge (\neg(x < n))) \rightarrow (x = n)$$

Figure 7. A loop and its specifications.

As shown in Figure 7, the knowledge base KB generates three verification conditions for the program. Given a candidate invariant I, the checker check the correctness of the verification conditions.

For example, a candidate invariant $\hat{I} \coloneqq (x < n)$ violates two verification conditions, correspondingly leading to two grounded rules.

For precondition correctness, the checker returns a counterexample $(x, n) = (0, 0)$. It reflects a grounded rule which should be satisfied by all invariants:

$$ce_1 : (0 = 0) \rightarrow I[x \mapsto 0, n \mapsto 0] \tag{7}$$

For inductive correctness, a counterexample is $(x, n, x') = (0, 1, 1)$ and the corresponding rule is

$$ce_2 : ((0 < 1) \wedge I[x \mapsto 0, n \mapsto 1] \wedge (1 = 0 + 1)) \rightarrow (I[x \mapsto 1, n \mapsto 1]) \tag{8}$$

The counterexamples are then collected in the corresponding sets as grounded rules CE to measure the consistency of a future candidate I

$$\texttt{Con}(CE, I) = \frac{1}{|CE|} \sum_{ce \in CE} \mathbb{I}(I \models ce). \tag{9}$$

A higher consistency value for a candidate indicates that it makes fewer mistakes made by other candidates.

B.2. Abduction Module and Abductive Learning

Algorithm 1 ABDUCTION

Input: Candidate invariant $\hat{I}$, program S
Parameter: Population N, iteration T, mutation rate δ
Output: Abduced Invariant $\tilde{I}$

Population $\leftarrow$ INITIALIZE-POPULATION($\hat{I}, N$)
$CE \leftarrow$ Empty-Set
Update CE by checking all *Population*
for $t = 1$ to T **do**
  New-Population $\leftarrow$ REPRODUCE(*Population*,δ) ▷ Perform RECOMBINE and MUTATE N times
  Update CE by checking all *New-Population*
  if A proper solution exists in *New-Population* **then**
    return BIASED-SELECT(*New-Population*) ▷ Return the simplest one among all verified solutions
  end if
  Population $\leftarrow$ SELECT(*New-Population* $\cup$ *Population*, CE) ▷ Select the next *Population* with $\texttt{Con}(CE; I)$
end for
return Abduction-Failure

Algorithm 2 LEARNING TO INFER LOOP INVARIANTS

Input: Unlabeled code files $\{S_i\} \sim \mathcal{S}$, initial model f_{θ_0}
Parameter: Number of iterations T
Output: Trained model f_{θ_t}

$\mathcal{D} \leftarrow$ Empty-Set
for $t = 1$ to T **do**
  $\mathcal{D}_t \leftarrow$ Empty-Set
  for S_i in $\{S_i\}$ **do**
    $\hat{I}_i \leftarrow f_{\theta_{t-1}}(S_i)$
    $\tilde{I}_i \leftarrow$ ABDUCTION $(\hat{I}_i, S_i)$ ▷ Revise $\hat{I}$ to make it consistent with KB
    if $\tilde{I}_i$ is not Abduction-Failure **then**
      Add $(S_i, \tilde{I}_i)$ to $\mathcal{D}_t$ ▷ Successfully abduced instances are used for training
    end if
  end for
  $\theta_t \leftarrow$TRAIN$(\theta_{t-1}, \mathcal{D}_t)$ ▷ Update model parameters so that its predictions are more likely to be consistent with KB
end for
return f_{θ_T}

C. Case Study

C.1. Comparison with State-of-the-art

In this section, we present empirical results from different generation methods. Eldarica and ImplCheck are based on different language bias and refinement policies and therefore, their performance differs. On one hand, when the language bias matches the encountered program, they can efficiently solve these problems. For example, Eldarica solves the problem in Figure 10 within 0.5 second. On the other hand, however, these methods may fail if the language bias is misaligned with the problem domain. For example, it takes Eldarica about 4 seconds in Figure 8, and the solver fails within the time limit if the termination condition is $x < 999999$. This situation makes them severely restricted when dealing with larger-scale programs or programs outside of distribution. Meanwhile, as ALIVE combines the code understanding ability of learning models with the logical reasoning ability of symbolic solvers, it can automatically learn a strategy to generalize. Empirical results show that the generated invariants are closer to those by human experts, which indicates that the model has learned to infer loop invariants by training with ALIVE.

```
int main() {
    // preconditions
    int x = 0;
    int y = unknown();
    assume( y % 6 == 0 );
    // loop
    while (x < 99) {
        if (y % 6 == 0) {
            x += 6;
        } else {
            x++;
        }
    }
    // postconditions
    assert((x % 6) == (y % 6));
    return 0;
}
```

Invariants generated by

- Eldarica: $((x = 102 \wedge y\%6 = 0) \vee (x = 96 \wedge y\%6 = 0)) \vee (x = 90 \wedge y\%6 = 0) \vee \cdots \vee (x = 0 \wedge y\%6 = 0)$ ✔
- ImplCheck: $[(-1) \times (y\%6) \geq 0] \wedge [(-1) \times (x\%6) \geq 0] \wedge [x + (y\%6) \geq 0]$ ✔
- Loopy (GPT-4): $(x\%6 = 0) \wedge (y\%6 = 0) \wedge (x \geq 99)$ ✔
- ALIVE (CodeT5): $x\%6 = 0 \wedge y\%6 = 0$ ✔
- Human Expert: $x\%6 = 0 \wedge y\%6 = 0$ ✔

Figure 8. Eldarica has to traverse all concrete states of variable x to synthesize the final invariant.

```
int main(){
    // preconditions
    int a,b,c;
    assume( a >= 0 );
    assume( a <= b );
    assume( b < c );
    // loop
    while( a < c ){
        a += 1;
        if(a > b) b += 1;
    }
    // postconditions
    assert( b == c );
    return 0;
}
```

Invariants generated by

- Eldarica: $(a = b \wedge c \geq a \wedge a \geq 0 \wedge c \geq 1) \vee (b \geq a \wedge (c - a) \geq 1 \wedge a \geq 0 \wedge (c - b) \geq 1)$ ✔
- ImplCheck: $(a \geq 0) \wedge (b + (-1) \times a \geq 0) \wedge (c + (-1) \times a \geq 0)$ ✔
- Loopy (GPT-4): $a \leq b \wedge b \leq c$ ✔
- ALIVE (CodeT5): $b \leq c \wedge b \geq a$ ✔
- Human Expert: $a \leq b \wedge b \leq c \wedge a \leq c$ ✔

Figure 9. The invariant generated by ALIVE (CodeT5) is more concise than the human expert.

```
int main() {
    // preconditions
    unsigned int v,x,h,d;
    h = 0;
    d = 0;
    x = 1;
    v = 0;
    // loop
    while (unknown()) {
        h++;
        d += (x - v);
        v += 18;
        x += 18;
    }
    // postconditions
    assert(h == d);
    return 0;
}
```

Invariants generated by

- Eldarica: $v - x = -1 \wedge h = d \wedge v \geq 0 \wedge h \geq 0$ ✔
- ImplCheck: $(((v + ((-1) \times x)) \geq (-1)) \wedge ((h + d) \geq 0) \wedge ((x + ((-18) \times h)) \geq 1) \wedge ((x + ((-2) \times v) + (18 \times h)) \geq 1) \wedge ((x + ((-18) \times d)) \geq 1) \wedge ((x + ((-2) \times v) + (18 \times d)) \geq 1))$ ✔
- Loopy (GPT-4): $d = h \times 18 - 17$ ✘
- ALIVE (CodeT5): $h = d \wedge x - v = 1$ ✔
- Human Expert: $h = d \wedge x - v = 1$ ✔

Figure 10. ImplCheck's answer is correct but verbose in this case.

```
int main(){
    // preconditions
    int i, n, sum;
    assume(n >10);
    i = 0;
    sum = 0;
    // loop
    while(i < n)
    {
        i = i + 1;
        sum = sum + i;
    }
    // postconditions
    assert(2*sum == n*(n+1));
    return 0;
}
```

Invariants generated by

- Eldarica: TIMEOUT ✘
- ImplCheck: TIMEOUT ✘
- Loopy (GPT-4): $sum = i \times (i + 1) \div 2$ ✘
- ALIVE (CodeT5): $i \leq n \wedge 2 \times sum = (i + 1) \times i$ ✔
- Human Expert: $i \leq n \wedge sum = i \times (i + 1) \div 2$ ✔

Figure 11. Eldarica and ImplCheck fail to generate a proper invariant within 500 seconds.

```
int main(){
    // preconditions
    int i,n,k,flag;
    i = 0;
    assume( n > 0 && n < 10 );
    assume( k > n - 2000 );
    assume(flag == 1 || flag == 0);
    // loop
    while( i < n ){
        i += 3;
        if(flag)
            k += 12000;
        else
            k += 2000;
    }
    // postconditions
    assert( k > n );
    return 0;
}
```

Figure 12. An initially unlabeled instance in the training set.

C.2. The Role of Learning Models in the Training Stage

Figure 12, Table 5 and Table 6 show the source code, output in the training stage and Jaccard distance between different pseudo labels respectively. In the first 6 iterations, no pseudo labels are used for training in this instance. However, the model prediction gradually gets closer to the human annotation, i.e., the Jaccard similarity between $\hat{I}$ and I gets larger. From this perspective, the model is a *student* as all supervision comes from the abduction module. In the 6th iteration, the first pseudo label is found. The Jaccard similarity between the model prediction and abduced result is 0.33, which indicates that the learning model provides a good initial solution so that the abduction process is facilitated by the *teacher*. Therefore, the interaction between the learning model and the abduction module is characterized by mutual benefit, where each component supports and enhances the other, leading to a more effective and adaptive training process.

Iteration	Model Prediction $\hat{I}$	Abduced Result $\tilde{I}$
1	$n > k + 12000$	Fail
2	$n > (k\%12000) \wedge k \geq 12000$	Fail
3	$n > (k\%12000) \wedge k \geq n$	Fail
4	$n > k \wedge 0 \leq i$	Fail
5	$n > k \wedge i < n$	Fail
6	$(i < n \vee k > n) \wedge (k < n \vee k > n)$	$(i < n \vee k > n) \wedge (10 < n \vee k > n - 2000) \wedge n \leq 10$
7	$(i < n \vee k > n) \wedge (10 < n \vee k > n - 2000) \wedge n \leq 10$	$(i < n \vee k > n) \wedge (10 < n \vee k > n - 2000) \wedge n \leq 10$
8	$(i < n \vee k > n) \wedge n < k + 2000$	$(i < n \vee k > n) \wedge n < k + 2000$

Table 5. Output of the learning model and the abduction module in the training stage.

Iteration	Jaccard$(\hat{I}, \tilde{I})$	Jaccard$(\hat{I}, I)$	Jaccard$(\tilde{I}, I)$
1	-	0.00	-
2	-	0.00	-
3	-	0.00	-
4	-	0.00	-
5	-	0.25	-
6	0.33	0.50	0.60
7	1.00	0.60	0.60
8	1.00	1.00	1.00

Table 6. Jaccard distance between different pseudo labels. Here I is labeled by the human expert and is only used for evaluation.
$I \coloneqq (i < n \wedge (k > n - 2000)) \vee k > n$