Interval Counterexamples for Loop Invariant Learning

Anonymous Author(s)

ABSTRACT

Loop invariant generation has long been a challenging problem. The black-box learning has recently emerged as a promising method for inferring loop invariants. However, the performance depends heavily on the quality of collected examples. In many cases, only after tens or even hundreds of constraint queries, can a feasible invariant be successfully inferred.

To reduce the gigantic number of constraint queries and improve the performance of black-box learning, we introduce interval counterexamples into the learning framework. Each interval counterexample represents a set of counterexamples from constraint solvers. We propose three different generalization techniques to compute interval counterexamples. The existing decision tree algorithm is also improved to adapt interval counterexamples. We evaluate our techniques and report over 40% improvement on learning rounds and verification time over the state-of-the-art approach.

CCS CONCEPTS

• **Theory of computation → Logic and verification**; • **Software and its engineering → Formal software verification**.

KEYWORDS

Program verification, invariant inference, constraint solving, decision tree learning, counterexample generalization

ACM Reference Format:

Anonymous Author(s). 2020. Interval Counterexamples for Loop Invariant Learning. In *Proceedings of The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Proving the correctness of loops has long been a challenging verification problem. Given a precondition, a postcondition, and a while loop, we would like to verify whether the postcondition always holds after the loop is executed from every program state satisfying the precondition. In order to prove the correctness of loops, one can find loop invariants to summarize the computation of loop iterations. Informally, a loop invariant is a predicate characterizing all program states at the loop head during executions. If a loop invariant furthermore includes program states admitted by the initial precondition and excludes states deviating from the postcondition, the correctness of the loop is established. Our goal is therefore to find *feasible* loop invariants for the loop with respect to the given pre- and post-conditions.

More specifically, we focus on the black-box *learning* model for loop invariant generation [10, 13, 14] in this paper. Intuitively, a learning model consists of two components: a white-box *teacher* (i.e. the verifier) and a black-box *learner*. The learner is completely agnostic of the program and the specification. Only sampled program states are revealed to the learner. Based on this limited information, the learner proposes an invariant to the teacher. If the purported invariant successfully establishes the correctness of the loop, we are done. Otherwise, more examples are collected and sent to the learner to refine its purported invariants. The process repeats until a feasible loop invariant or a bug is found.

Decision tree inference algorithms have been proposed [15, 19] as the learner in the black-box learning framework. Recall that loop invariants are indeed predicates on program states. Decision trees can also be seen as binary classifiers on program states. From the collected examples, an algorithm can hence generate decision trees as loop invariants. More specifically, an example is classified *true* or *false* depending on if it is accepted by the invariant or not. Garg et al. [14] further argue that merely learning from *true* and *false* examples for invariant synthesis is inherently non-robust. They propose a robust learning model, with Implication Counterexamples and Examples (*ICE*).

The performance of the learning framework depends heavily on the quality of collected examples. When complete information is collected, the loop invariant can surely be inferred. It however requires a gigantic amount of examples. Currently, the examples are mainly obtained from constraint solvers: a model of a satisfiable query corresponds to a program state. With sufficiently many constraint queries, a number of examples are collected. Decision tree inference algorithms can then infer feasible loop invariants. In practice, tens or even hundreds of constraint queries are made before loop invariants are found. Since constructing decision trees and constraint solving can be expensive, reducing the number of queries can improve the performance of such verification algorithms.

On examining models returned by constraint solvers, we find that the models from constraint solvers are very *specialized* and *unpredictable*. Consider an *unknown* predicate $x > 0$ where x is an integer variable. Any positive integer can be a model (called a *point model*) for the constraint. A constraint solver may return $x = 1000, 999, 998, \ldots$ as models of the query. Subsequently, a decision tree inference algorithm needs lots of examples before it infers the predicate $x > 0$. Yet there is no essential difference among these examples. If a constraint solver could return a so-called *interval model*, say $x > 0$, in one query, the decision tree algorithm would immediately infer the predicate. Generalized interval models from constraint solving may greatly improve the performance of loop invariant inference in the black-box learning framework.

In this paper, we propose three techniques to obtain generalized interval models from constraint solvers and apply them to loop invariant inference. By novel applications of UNSAT core computation in constraint solvers, our techniques reduce the number of learning rounds up to 42%. In order to incorporate generalized models from constraint solvers, we moreover propose a decision tree inference algorithm with interval examples for loop invariant inference. With our new techniques, the total verification time is

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

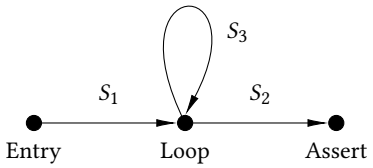


Figure 1: A simple loop program

reduced up to 47% for 94 benchmarks from prior work and software verification competition.

Our contributions are summarized as follows.

(1) We introduce interval counterexamples which represent a set of counterexamples from constraint solvers.
(2) We propose three different generalization techniques to compute interval counterexamples.
(3) We improve the existing decision tree algorithm to adapt interval counterexamples.
(4) We evaluate our techniques and report over 40% improvement on learning rounds and verification time.

The paper is organized as follows. We review backgrounds in Section 2. Our counterexample generalization techniques will be presented in Section 3. Section 4 develops our decision tree inference algorithm with interval examples. It is followed by experiments and evaluation in Section 5. Section 6 briefly reviews related work. We conclude the presentation in Section 7.

2 BACKGROUND

2.1 Black-box Loop Invariant Inference

The black-box method for loop invariant inference [10, 13, 14, 35] consists of a learner and a teacher. The learner tries to find a loop invariant guided by the teacher without examining details of the loop body under verification. Rather, only sampled program states at the loop head are revealed to the learner. Based on the limited information, the learner proposes an invariant to the teacher. The teacher, on the other hand, tries to verify the correctness of the loop with the proposed invariant. If the verification succeeds, a feasible loop invariant is found. Otherwise, the teacher must find a counterexample to explain why the invariant fails. There are two scenarios. The counterexample may reflect a program execution that rejects the assertion. The teacher then reports that the program is incorrect. Otherwise, the counterexample is sent to the learner to improve the proposed invariant.

To simplify our presentation, we will consider programs of the following form throughout the paper:

assume(Φ_{pre}); S_1; **while**(*cond*) **do** S_3 **od** S_2; **assert**(Φ_{post});

where Φ_{pre} and Φ_{post} are pre- and post-conditions of the program, respectively; S_1, S_2 and S_3 are sequences of program statements before, after, and in the loop, respectively; and *cond* is the loop condition. Its control flow graph is shown in Figure 1.

Let $\vec{x}$ be the set of variables occurred in the program. In order to denote values of a variable at different program locations, we use $\vec{x}_i$'s to denote copies of $\vec{x}$ at different locations. A (program) *state* at a location thus is a valuation on $\vec{x}_i$. A statement sequence moreover can be seen as a *transformation relation* between different copies of program variables. In Figure 1, we use $\rho_1(\vec{x}, \vec{x}')$, $\rho_2(\vec{x}, \vec{x}')$ and $\rho_3(\vec{x}, \vec{x}')$ to denote the transformation relations of the statement sequences S_1, S_2 and S_3, respectively.

A *loop invariant* $\phi(\vec{x})$ with respect to the precondition $\Phi_{pre}(\vec{x})$ and postcondition $\Phi_{post}(\vec{x})$ satisfies the following three formulas:

$$\psi_1 : \Phi_{pre}(\vec{x}_0) \wedge \rho_1(\vec{x}_0, \vec{x}_1) \rightarrow \phi(\vec{x}_1), \quad (1)$$

$$\psi_2 : \phi(\vec{x}_2) \wedge \neg cond(\vec{x}_2) \wedge \rho_2(\vec{x}_2, \vec{x}_3) \rightarrow \Phi_{post}(\vec{x}_3), \quad (2)$$

$$\psi_3 : \phi(\vec{x}_2) \wedge cond(\vec{x}_2) \wedge \rho_3(\vec{x}_2, \vec{x}_4) \rightarrow \phi(\vec{x}_4), \quad (3)$$

ψ_1 states that the loop invariant must include the program states resulted from initial states satisfying the pre-condition. ψ_2 specifies that the post-condition must hold when leaving the loop. ψ_3 says that the invariant holds after each iteration. We write Ψ for $\psi_1 \wedge \psi_2 \wedge \psi_3$ for simplicity.

Note that the loop invariant ϕ in the above formulas is a placeholder. Let H be a *hypothetical* loop invariant. $\Psi(H)$ denotes the formula obtained by replacing ψ with H in Ψ. The *loop invariant inference problem* is to find an H such that $\Psi(H)$ is valid.

The validity of $\Psi(H)$ can be checked by constraint solvers. If $\Psi(H)$ is valid, we call the hypothesis H a *feasible* loop invariant. Otherwise, the solver returns a model σ that falsifies $\Psi(H)$. Note that σ is a valuation on $\vec{x}_i$ ($1 \leq i \leq 4$) in Ψ. Denote $\sigma(x_i)$ the valuation of σ on $\vec{x}_i$. Since $\Phi(H)$ is a conjunction, at least one of the formulas $\psi_1(H)$, $\psi_2(H)$, or $\psi_3(H)$ must be false. If $\psi_1(H)$ is false, the purported invariant H should but does not include the state $\sigma(\vec{x}_1)$. We hence call $\sigma(\vec{x}_1)$ a *positive* counterexample. If $\psi_2(H)$ is false, H should not but includes the state $\sigma(\vec{x}_2)$. We get a *negative* counterexample. If $\psi_3(H)$ is false, we get an *implication* counterexample $(\sigma(\vec{x}_2), \sigma(\vec{x}_4))$ [14]. Informally, the state $\sigma(\vec{x}_4)$ should be included by the purported invariant H if $\sigma(\vec{x}_2)$ is. Note counterexamples only contains states sampled at the loop head.

When a state is both a positive and negative counterexample, it reflects a real bug in the loop program, since it is reachable from the initial state (by positive counterexample) and also fails the postcondition (by negative counterexample).

A high-level view for the black-box loop invariant inference framework is shown in Algorithm 1. In the algorithm, Γ is the counterexample set and initially empty. H is the hypothetical loop invariant purported by the learner and initially $\top$. The algorithm continues if the teacher finds a counterexample (line 2) and give counterexamples to the learner to improve the hypotheses (lines 6-7). If the counterexample is a bug (line 3), the algorithm reports a failure. If no counterexample can be found, the algorithm returns H as a feasible loop invariant for verification.

2.2 Inference with Decision Trees

Decision tree inference [30, 31] is a method commonly used in data mining and machine learning. It has been proposed in [15, 19] as the learner in the black-box loop invariant learning framework. Garg et al. [14] argue that learning using examples only does not form a robust learning framework for invariant synthesis. They propose a robust learning model, called *ICE-learning*, that extends the classical learning framework with implications.

In the ICE framework, a sample to the learner is a triple of three sets, i.e., $\mathcal{S} = \langle \mathcal{S}^+, \mathcal{S}^-, \mathcal{S}^U \rangle$, where $\mathcal{S}^+$ is the set of examples with

Algorithm 1: Black-box learning framework

```
1  Γ = ∅; H = ⊤;
2  while ∃σ.σ ⊨ ¬Ψ(H) do
3      if σ is a real counterexample then
4          return assertion failed;
5      end
6      Γ = Γ ∪ σ;
7      H = Learner(Γ);
8  end
9  return H;
```

the label *true* (produced by the *positive* counterexamples), $\mathcal{S}^-$ is the set of examples with the label *false* (produced by the *negative* counterexamples), and $\mathcal{S}^U$ is the set of examples that have not been classified (produced by end-points of the implication counterexamples). The ICE learner's job is to classify all *unclassified* examples without breaking their implication relation, and to separate the *true* examples and *false* examples.

Let $\mathcal{A}$ be a set of attributes, i.e., certain arithmetic combinations of integer variables. During the construction of the decision tree, each internal node represents a predicate of the form $a_k \le t$, where a_k is an attribute and t is a constant, each branch denotes a test on the predicate, and each leaf (or terminal) node is labeled *true* or *false*. The finalized decision tree represents a Boolean combination of predicates.

Given a sample $\mathcal{S}$ and attributes $\mathcal{A}$, the ICE learning algorithm (Algorithm 2) constructs the decision tree in a top-down fashion. At each stage of the construction (line 9), it chooses an attribute $a_k \in \mathcal{A}$ and a threshold t such that the composed predicate $a_k \le t$ "best" (will be explained soon) classifies the sample. Denote $\mathcal{S}_{a_k \le t}$, $\mathcal{S}_{a_k > t}$ the split results, i.e., those that satisfy the predicate and those that do not satisfy it. The algorithm then recurses on these two sub-samples (line 10 to 11) and builds two subtrees. If either $\mathcal{S}^+$ or $\mathcal{S}^-$ is empty (line 3), indicating that the classification is complete, all unclassified examples in $\mathcal{S}^U$ are marked using the same label of classified examples. After an implication example been classified, its label should be propagated [15] (with respect to the implication relation) to as many other unclassified examples as possible. The propagated examples are recorded in a global data structure G (at line 6) to be shared to other call to the algorithm.

The crucial step for the decision tree learning algorithm is how choosing an attribute a_k and a threshold t that "best" classify the sample. Most learning algorithms (e.g., ID3 [28], C4.5 [29], ICE [14]) use *Shannon entropy* [32] to measure the impurity of a sample, i.e.,

$$Entropy(\mathcal{S}) = -\frac{p}{p+n} log_2 \frac{p}{p+n} - \frac{n}{p+n} log_2 \frac{n}{p+n},$$

where p and n are numbers of *true* and *false* examples in the sample, respectively. A lower entropy indicates that this sample has been classified more complete, and hence is more preferred. The *information gain* is used to measure the split of $\mathcal{S}$ into $\mathcal{S}_{a_k \le t}$ and $\mathcal{S}_{a_k > t}$, i.e.,

$$Gain(\mathcal{S}, \mathcal{S}_{a_k \le t}, \mathcal{S}_{a_k > t}) = Entropy(\mathcal{S}) - Entropy(\mathcal{S}_{a_k \le t}) - Entropy(\mathcal{S}_{a_k > t})$$

Algorithm 2: Decision tree learning with implication counterexamples [15]

```
Data: an implication relation impl, and a partial valuation G
      of unclassified examples in impl
1  Procedure ICE-DT(S = ⟨S+, S−, SU⟩, A)
2      Move all classified examples in G to S+ and S−,
       respectively ;
3      if S− = ∅ or S+ = ∅ then
4          Let label be true or false, respectively;
5          Mark all examples in SU with label ;
6          G ← Propagate(SU, impl);
7          return a single node tree Root, with label ;
8      else
9          Select an attribute ak and a threshold t that best
           classify S ;
10         Split S into two parts S_{ak≤t} and S_{ak>t};
11         T0 ← ICE-DT(S_{ak≤t}, A);
12         T1 ← ICE-DT(S_{ak>t}, A);
13         return a tree with root (labeled with ak ≤ t), left
           subtree T0 and right subtree T1.
14     end
15 end
```

We prefer the pair of attribute and threshold that achieves the most information gain. Moreover, Garg et al. [15] observed that the two examples of an implication should be classified as the same class. They thus suggest to penalize the split that *cuts* implications (i.e., split two examples of an implication to different parts) in the above information gain formula.

Lemma 2.1. *[15] The ICE algorithm, independent of how the attributes and threshold are chosen for the split of the sample, always terminates and produces a decision tree that is consistent with the input sample.*

3 COUNTEREXAMPLE GENERALIZATION

In this section, we motivate the improvement for the existing black-box loop invariant inference framework. Three counterexample generalization methods based on UNSAT cores are then proposed.

3.1 Motivation

In the loop invariant inference framework (Algorithm 1), we only get one program state as a counterexample in each round. Intuitively, the amount of information collected in each round is very limited. Although the constraint solver only returns one valuation each time, there may be many similar valuations satisfying the same constraint. We would like to explore as many valuations for decision tree inference as possible. By observing counterexamples from constraint solvers, we find that the standard framework repeatedly finds counterexamples from the same control flow path. To reduce repetition, we wish to generalize each counterexample along its corresponding control flow path. After generalization, each variable has an interval range. Every valuation in the intervals

```
1  int x, y;
2  assume(x <= 2);
3  assume(y >= 0 && y <= 1);
4  while (*){
5    if (x > 0) {
6      x := x - 1;
7      y := y + 1;
8    }
9  }
10 assert(y >= 0 && y <= 3);
```

Figure 2: A simple program example

will be a counterexample. Subsequently, we call these generalized counterexamples by *interval* counterexamples.

Example 3.1. We illustrate our ideas with the program in Figure 2. Initially, the invariant $H_0 : \top$ is proposed. According to Algorithm 1, the framework tries to find a counterexample to falsify $\Phi(H_0)$. The constraint solver may return $\langle x = 0, y = 10\rangle$. Under this valuation, the program fails the assertion at line 10 and hence falsifies $\psi_2(H_0)$ (the formula (2)). The valuation should be excluded from the loop invariant. With the negative counterexample, the learner may propose $H_1 : y \leq 9$ in the next round. Since $\langle x = 0, y = 9\rangle$ falsifies $\psi_2(H_1)$, the teacher may return the new negative counterexample. The process repeats until the feasible loop invariant $-1 < y \leq 3 \wedge x + y \leq 3$. Notice that the counterexamples $\langle x = 0, y = 10\rangle$ and $\langle x = 0, y = 9\rangle$ correspond to the same control flow path (leaving the loop immediately). Our generalization aims to reduce such repetition. If the teacher could return an interval counterexample $\langle x \in [0, 0], y \in [4, \infty)\rangle$, the efficiency would be greatly improved.

3.2 Generalization Framework

Based on the motivation, we introduce counterexample generalization into the loop invariant inference framework. The goal of generalization is to find more program states at the loop head as counterexamples to improve loop invariants. The new framework is shown in Algorithm 3. Compared to Algorithm 1, there are two modifications. At line 6, the counterexample σ is generalized before putting into the counterexample set Γ. At line 7, the classical decision tree learner is extended to support interval examples (Section 4).

Let σ be a valuation falsifying $\Phi(H)$ for a purported invariant H. A program state (called a *seed state*) at the loop head is obtained from σ. From a seed state, a set of states (called *generalized states*) is obtained. Generalized states must also be counterexamples to the hypothetical invariant H. Specifically, if σ falsifies ψ_1, the seed state $\sigma(\vec{x}_1)$ is a positive counterexample. We want to find more valuations on $\vec{x}_1$ falsifying ψ_1. These valuations are the generalized states to $\sigma(\vec{x}_1)$. Formally, consider the set V^+ of valuations falsifying the formula

$$\exists \vec{x}_0.\, \Phi_{pre}(\vec{x}_0) \wedge \rho_1(\vec{x}_0, \vec{x}_1) \rightarrow \phi(\vec{x}_1). \tag{4}$$

Algorithm 3: Extended learning framework with interval generalization

```
1 Γ = ∅; H = ⊤;
2 while ∃σ.σ ⊨ ¬Ψ(H) do
3   if σ is a real counterexample then
4     return assertion failed;
5   end
6   Γ = Γ ∪ Generalize(σ);
7   H = IntervalLearner(Γ);
8 end
9 return H;
```

Note $\sigma(\vec{x}_1) \in V^+$. All valuations in V^+ are positive counterexamples.

Similarly, if σ falsifies ψ_2, the state $\sigma(\vec{x}_2)$ is a negative counterexample. We compute the set V^- of valuations falsifying the formula

$$\exists \vec{x}_3.\, \phi(\vec{x}_2) \wedge \neg cond(\vec{x}_2) \wedge \rho_2(\vec{x}_2, \vec{x}_3) \rightarrow \Phi_{post}(\vec{x}_3). \tag{5}$$

Apparently, $\sigma(\vec{x}_2) \in V^-$. All valuations in V^- are negative examples. Note that in the above generalization formulas (4) and (5), the transformation relations (ρ_1 or ρ_2) may contain other intermediate variables. In that case, these intermediate variables should also be existentially quantified.

3.3 Path-Driven Quantifier Elimination

To eliminate the existential quantifiers in (4) and (5), we use the loop-head variables to substitute all other variables. More concretely, we limit the generalization along one control flow path in the program fragment (S_1 or S_2). Recall that the counterexample σ is a valuation on all variables in Ψ. If σ falsifies $\psi_1(H)$, the program states $\sigma(\vec{x}_0)$ and $\sigma(\vec{x}_1)$ expose a program path from the entry location to the loop head. We denote this path by π_1. Similarly, if σ falsifies $\psi_2(H)$, $\sigma(\vec{x}_2)$ and $\sigma(\vec{x}_3)$ expose a program path from the loop head to the assert location. We denote it by π_2.

Given the path π_1, we collect all conditions along the path and use their conjunction to replace the SSA encoding ρ_1 in $\psi_1(H)$. Similarly, we replace the SSA encoding ρ_2 in $\psi_2(H)$ with the conjunction of conditions along π_2. Each `assign` statement on the path is treated as a substitution. Specifically, we use $\vec{x}_1$ (at the end of π_1) to replace all other variables in $\psi_1(H)$ for each `assign` statement along π_1. Similarly, we use $\vec{x}_2$ (at the beginning of π_2) to substitute all variables in $\psi_2(H)$ along π_2 in the *reverse* order. After substitution, we obtain *generalization* formulas $\psi_1^\sharp(H)$ and $\psi_2^\sharp(H)$ from (4) and (5) respectively. Both formulas are quantifier-free and contain only variables at the loop head.

Example 3.2. Let us assume both π_1 and π_2 contain two `assign` statements, i.e., "$x_1 = x_0 + 1;\ x_3 = x_1$". These two statements on π_1 correspond to the substitutions $x_0 \mapsto x_1 - 1$ and $x_1 \mapsto x_3$; and them on π_2 correspond to the substitutions $x_3 \mapsto x_1$ and $x_1 \mapsto x_0 + 1$. Note that for the path on π_2, the substitutions need to be applied in the reverse order. Moreover, if there is an `assign` statement where the right operand uses more than one variable, e.g., "$y_2 = x_3 + z_2$", we cannot use "$y_2 - z_2$" to substitute "x_3" (because z_2 needs also to

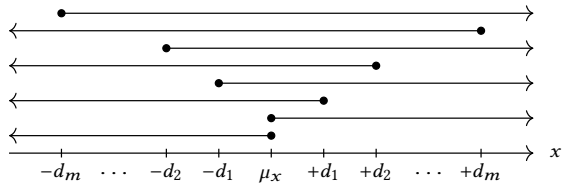


Figure 3: Range Generalization

be removed). This problem can be solved by replacing z_2 with its concrete value in σ.

3.4 Generalization by Variable Elimination

Let ψ be a generalization formula $\psi_1^{\sharp}(H)$ or $\psi_2^{\sharp}(H)$, $\vec{x}$ the set of variables in ψ, and σ a valuation falsifying ψ. The program state $\sigma(\vec{x})$ (at the loop head) is hence a positive or negative counterexample.

One can generalize $\sigma(\vec{x})$ by computing the UNSAT core as follows. Denote $\sigma(x)$ by μ_x for $x \in \vec{x}$. The seed state $\sigma(\vec{x})$ can be represented by the formula $\bigwedge_{x\in\vec{x}} x = \mu_x$. Recall that $\sigma(\vec{x})$ falsifies ψ. The following formula is hence unsatisfiable:

$$\psi \wedge \bigwedge_{x\in\vec{x}} x = \mu_x \tag{6}$$

Using ψ as the hard constraint and equalities $x = \mu_x$ for $x \in \vec{x}$ as soft constraints, an UNSAT core of the formula (6) gives a subset U of the soft constraints such that $\psi \wedge \bigwedge\{x = \mu_x | x = \mu_x \in U\}$ is still unsatisfiable. Denote $\vec{x}_c \subseteq \vec{x}$ the set of variables in the UNSAT core. A variable not in $\vec{x}_c$ means its value is inessential to the unsatisfiability of the formula (6). By eliminating μ_x for $x \notin \vec{x}_c$, generalized states are obtained from the seed state $\sigma(\vec{x})$. Intuitively, generalized states are obtained by taking an arbitrary value for each variable not in the UNSAT core.

Example 3.3. In our motivating example, we find a negative counterexample $\sigma : \langle x = 0, y = 10\rangle$ from the initial invariant $H : \top$. Note that the statement sequence after the loop and before the assertion in the program is empty. By (5), the formula ψ is $\top \wedge \top \wedge \top \rightarrow y \geq 0 \wedge y \leq 3$, and $y \geq 0 \wedge y \leq 3$ after simplification. By (6), the formula is $\psi \wedge x = 0 \wedge y = 10$. Observe that the above formula is still not satisfiable after removing $x = 0$. We thus obtain generalized states where $x \in (-\infty, +\infty)$ and $y = 10$. All of them are negative counterexamples.

3.5 Generalization by Boundary Constraints

Generalization by variable elimination however is too coarse in practice. When a variable is removed from the seed state, its value is irrelevant to the counterexample. This is possible but unlikely. In most cases, a variable is not removable but can be generalized to a smaller range. In order to restrict ranges of generalization, we propose the *boundary* generalization technique.

Let $\mathcal{D} = \{d_0, d_1, d_2, \ldots, d_m\}$ be a sequence of distances with $d_0 = 0$ and $d_j < d_{j+1}$ for every $0 \leq j < m$. We would like to use $\mathcal{D}$ to determine the boundaries of generalization ranges incrementally. Specifically, we generalize the value μ_x for the variable x by $2m+2$ boundary inequalities $\{x \geq \mu_x - d_i, x \leq \mu_x + d_j \,|\, 0 \leq i, j \leq m\}$. Using ψ as the hard constraint and these boundary inequalities as soft constraints, we compute an UNSAT core of the following formula:

$$\psi \wedge \bigwedge_{x\in\vec{x}} \bigwedge \{x \geq \mu_x - d_i, x \leq \mu_x + d_j \,|\, 0 \leq i, j \leq m\} \tag{7}$$

Each inequality of the soft constraint in (7) corresponds to a half-bounded interval $[\mu_x - d_i, +\infty)$ or $(-\infty, \mu_x + d_j]$ (Figure 3). Intuitively, an UNSAT core gives us a subset of such intervals.

If the equality $x = \mu_x$ cannot be generalized, the UNSAT core must contain $x \geq \mu_x$ and $x \leq \mu_x$. If the variable x can be generalized to any value in its domain, the UNSAT core contains no inequalities of x. Generalization by boundary constraints is therefore more general than variable elimination.

From the boundary inequalities in the UNSAT core, we compute the greatest left boundary lb and the least right boundary rb. The value μ_x for the variable x is generalized to $x \in [lb, rb]$ for each $x \in \vec{x}$. We thus obtain an *interval counterexample* by generalization. Note that the interval counterexample does not necessarily be symmetric to the value μ_x for the variable x.

Example 3.4. In our motivating example, recall the formula $\psi : y \geq 0 \wedge y \leq 3$ and the negative counterexample $\sigma : \langle x = 0, y = 10\rangle$. Choose $\mathcal{D} = \{0, 1, \ldots, 10\}$. By (7), we have

$$\psi \wedge \bigwedge_{0\leq i,j\leq 10} \{x \geq 0 - i, x \leq 0 + j\} \wedge \bigwedge_{0\leq i,j\leq 10} \{y \geq 10 - i, y \leq 10 + j\}.$$

The UNSAT core computation returns $\{y \geq 4\}$, i.e., x was eliminated, and y was generalized to $[4, +\infty)$. The interval counterexample is thus $x \in (-\infty, +\infty), y \in [4, +\infty)$.

Theorem 3.5. *Let $[lb, rb]$ be the interval counterexample obtained from the UNSAT core of the formula (7), any state satisfying $[lb, rb]$ falsifies ψ.*

3.6 Generalization by Interval Digging

Recall that UNSAT cores aim to minimize the number of constraints. It does not necessarily ensure the most general interval of each variable. Recall the generalization formula in Example 3.4, both $\{y \geq 4\}$ and $\{y \geq 5\}$ are its UNSAT cores. They both have only one constraint and are hence minimal. Observe that $y \geq 5 \Rightarrow y \geq 4$. To maximize the interval range, the UNSAT core $\{y \geq 4\}$ is of course preferred. Computing minimal UNSAT cores may not yield the most general interval for y.

In order to find the most general intervals among minimal UNSAT cores, we propose the *digging generalization*. This technique ensures that the soft constraints are pairwisely disjoint, and minimal UNSAT cores always lead to maximal generalization ranges.

Let $x \in \vec{x}$, and μ_x its value in a seed state. Similar to the previous technique, we also use the distance sequence $\mathcal{D}$ to incrementally determine the boundaries of the generalization range. The difference lies in the way how we specify the constraints. As shown in Figure 4, the domain of x (excluding μ_x) is the union of the following $2m+2$ intervals:

$$\mathcal{I}_{x,\mu_x,k} := \begin{cases} x < \mu_x - d_m, & \text{if } k = -\infty \\ x \geq \mu_x - d_{|k|} \wedge x < \mu_x - d_{|k|-1}, & \text{if } -m \leq k \leq -1 \\ x > \mu_x + d_{k-1} \wedge x \leq \mu_x + d_k, & \text{if } 1 \leq k \leq m \\ x > \mu_x + d_m, & \text{if } k = \infty \end{cases}$$

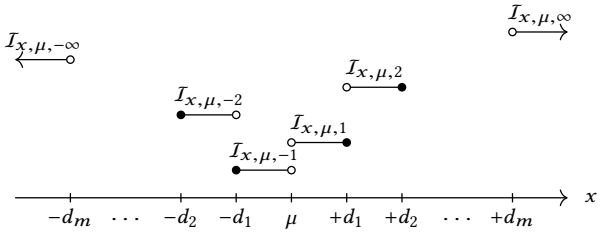


Figure 4: Interval Digging Generalization

Note that $\mathcal{I}_{x,\mu_x,k} \not\Rightarrow \mathcal{I}_{x,\mu_x,k'}$ when $k \neq k'$. The problem in boundary constraint generalization is thus prevented. Now consider the following query

$$\psi \wedge \bigwedge_{x \in \vec{x}} \bigwedge \{\neg \mathcal{I}_{x,\mu_x,k} \mid k = -\infty, -m, \ldots, -1, 1, \ldots, m, \infty\} \quad (8)$$

with the hard constraint ψ and soft constraints $\mathcal{I}_{x,\mu_x,k}$. Its unsat core gives a subset of intervals that should be excluded from the generalization of x.

For an UNSAT core $\{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_m\}$, note that $\bigwedge_{i=1}^{m} \neg \mathcal{I}_i$ may represent several disjoint intervals for a variable $x \in \vec{x}$. We choose the interval containing μ_x as the interval counterexample.

Theorem 3.6. *Let $\{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_m\}$ be the UNSAT core of the formula (8), any state satisfying $\bigwedge_{i=1}^{m} \neg \mathcal{I}_i$ falsifies ψ.*

4 INTERVAL DECISION TREE LEARNING

The classical decision tree learning framework deals with point examples only. This section presents our new decision tree learning algorithm that extends the classical learning model with interval examples.

Our learning algorithm builds on the ICE framework. The high-level description of our algorithm is *identical* to that of the ICE algorithm (in Algorithm 2). However, many low-level mechanisms need to be adapted to the interval setting.

4.1 Interval Examples

Let $\mathcal{A} = \{a_0, a_1, \ldots, a_{|\mathcal{A}|-1}\}$ be the set of attributes. A *point example* v is a valuation that assigns a value to each a_k in $\mathcal{A}$. An *interval example* I (produced by an interval counterexample) is a mapping that assigns a *closed* interval to each a_k in $\mathcal{A}$. In the following, we use $v(a_k)$ to refer to the value of v at a_k, and $I(a_k)$ to refer to the interval of I at a_k. We also use $I(a_k).lb$ and $I(a_k).rb$ to refer to the left and right boundaries of $I(a_k)$, respectively.

An interval example can be denoted in the constraint form as

$$I = \bigwedge_{a_k \in \mathcal{A}} I(a_k).lb \leq a_k \leq I(a_k).rb$$

We say that a point example v is *covered* by an interval example I if v (as a valuation) is a model of I (as a constraint).

Example 4.1. Consider the interval examples in Table 1, where each row corresponds to an example, and each column stands for an attribute. Observe that the first interval example covers a single point example, while the second interval example covers $5 \times 4 \times 1 = 20$ point examples.

Table 1: Interval examples

x	y	z	$x + y$	$x - y$
$[4, 4]$	$[1, 1]$	$[1, 1]$	$[5, 5]$	$[3, 3]$
$[2, 6]$	$[1, 4]$	$[2, 2]$	$[3, 10]$	$[-2, 5]$

4.2 Extending Entropy to Interval Examples

The existing mechanism for computing Shannon entropy applies to point examples only. We extend it to interval examples.

Formally, let $\mathcal{S} = \langle \mathcal{S}^+, \mathcal{S}^-, \mathcal{S}^U \rangle$ be an interval sample. For each interval example I in the sample, we compute its *weight* as

$$weight(I) = \min \left\{ \sqrt{\prod_{a_k \in \mathcal{A}} (I(a_k).rb - I(a_k).lb)}, MAX\_WEIGHT \right\},$$

where $I(a_k).rb, I(a_k).lb$ are boundaries of the interval $I(a_k)$, and MAX_WEIGHT is a value representing the maximum of weights. Note that the weight of any point example evaluates to 1. In the above definition, we use square roots and MAX_WEIGHT to balance the weight values between examples, thus to prevent a certain large-interval example dominates the whole learning process.

The *entropy* of an interval sample $\mathcal{S}$ is:

$$Entropy(\mathcal{S}) = -\frac{w^+}{w^+ + w^-} log_2 \frac{w^+}{w^+ + w^-} - \frac{w^-}{w^+ + w^-} log_2 \frac{w^-}{w^+ + w^-},$$

where w^+ and w^- are sum weights of examples in $\mathcal{S}^+$ and $\mathcal{S}^-$ of the sample, respectively. Note that the set $\mathcal{S}^U$ does not contribute to the entropy, since its examples are all unclassified. The above entropy can be used to measure the *impurity* of interval samples. Especially, the entropy of a sample is 0 if either $\mathcal{S}^+ = \emptyset$ or $\mathcal{S}^- = \emptyset$, and the entropy gets a high value if the sample contains nearly the same numbers of *true* and *false* examples.

4.3 Selecting Attributes and Thresholds with Interval Examples

Selecting an attribute and a threshold that best classify the sample is the most intricate and most time-consuming operation in the decision tree learning. To get the best solution, it often needs to calculate and compare the information gains of all pairs of candidate attributes and candidate thresholds. The situation is even worse after the introduction of interval samples, since with which the point set implied by the sample is greatly enlarged. Without specialized technique, searching in this large space is far from efficient.

As previously mentioned, an interval example covers a number of point examples. If an interval example is classified, all covered point examples are classified. Therefore, when split the sample, we prefer to not dividing the intervals. Formally, we say that an interval example I is *cut* by $\mathcal{S}_{a_k \leq t}$ and $\mathcal{S}_{a_k > t}$ if $I(a_k).lb \leq t < I(a_k).rb$. In this case, we also say that the split of $\mathcal{S}_{a_k \leq t}$ and $\mathcal{S}_{a_k > t}$ *cuts* the interval example, or that the predicate $a_k \leq t$ *cuts* the interval example. Especially, a split is *non-crossing* if it cuts no interval example in the sample.

To find the best split, we need to select an attribute and synthesize a threshold. In the case of point examples, with the information gain function (Section 2.2), the best threshold is always among the

values occurring in the points in the sample [29]. By regarding each interval example as a set of point examples, a similar conclusion can be drawn in our interval setting.

Lemma 4.2. *The best threshold for an attribute a_k is among the values in $\bigcup_{I\in\mathcal{S}} I(a_k)$, i.e., the union of intervals at a_k of all examples in the sample.*

Moreover, taking the nature of intervals into consideration, we have the following stronger conclusions.

Theorem 4.3. *The best threshold for an attribute a_k is among the values in $\mathbf{LB_{k,-1}} \cup \mathbf{RB_k}$, where $\mathbf{LB_{k,-1}} = \{I(a_k).lb - 1 \mid I \in \mathcal{S}\}$ and $\mathbf{RB_k} = \{I(a_k).rb \mid I \in \mathcal{S}\}$ are sets of values by respectively subtracting 1 from the left boundaries and taking the right boundaries of intervals at a_k of all examples in the sample.*

Proof. Assume there are n examples in the sample, and their intervals at a_k are $I_1(a_k), I_2(a_k), \cdots, I_n(a_k)$, respectively. Let t be an arbitrary threshold of a_k. The predicate $a_k \le t$ may cut several intervals, say $I_{i_1}(a_k), \cdots, I_{i_k}(a_k)$. Let $B = \{I_{i_1}(a_k).lb-1, I_{i_1}(a_k).rb, \cdots, I_{i_k}(a_k).lb-1, I_{i_k}(a_k).rb\}$ and $v \in B$ the *nearest* value to t. The value v is a better threshold than t. There is one interval that is cut by the predicate $a_k \le t$ but not by $a_k \le v$. In other words, for any threshold $t \notin \mathbf{LB_{k,-1}} \cup \mathbf{RB_k}$, there always exists a better one in $\mathbf{LB_{k,-1}} \cup \mathbf{RB_k}$. □

Theorem 4.4. *If non-crossing splits for an attribute a_k exist, the best non-crossing threshold for a_k is among the values in $\mathbf{RB_k} = \{I(a_k).rb \mid I \in \mathcal{S}\}$, i.e., the set of right boundaries of intervals at a_k of the examples in the sample.*

Proof. By premise, let t be a threshold that leads to a best non-crossing split on the sample. According to Theorem 4.3, the best threshold is among the values in $\mathbf{LB_{k,-1}} \cup \mathbf{RB_k}$. Assume $t \in \mathbf{LB_{k,-1}}$. Let $I_{i_1}(a_k), \ldots, I_{i_k}(a_k)$ be the set of intervals at the left side of the split (i.e., satisfy $a_k \le t$), and v the largest right boundary among these intervals. Setting v as the new threshold will not change the split result. In other words, for any threshold $t \in \mathbf{LB_{k,-1}}$, we can always find a threshold in $\mathbf{RB_k}$ that leads to the same classification result. Therefore, it is sufficient to search in $\mathbf{RB_k}$. □

According to Theorem 4.3 and Theorem 4.4, we develop the following strategies for selecting attributes and thresholds for interval examples:

(1) Find the *best non-crossing* split with the attribute $a_k \in \mathcal{A}$ and the threshold $t \in \mathbf{RB}_k$. Goto the next step if no non-crossing split exists.
(2) Find the *best crossing* split with the attribute $a_k \in \mathcal{A}$ and the threshold $t \in \mathbf{LB}_{k,-1} \cup \mathbf{RB}_k$.

4.4 Properties

According to Lemma 2.1, the ICE learning framework is correct, independent of how the attributes and thresholds are chosen to split the sample. Our adaptions of the ICE learning framework to interval examples exactly lie in this category, we therefore have the following deduction.

Theorem 4.5. *The interval decision tree algorithm always terminates and produces a decision tree that is consistent with the input sample.*

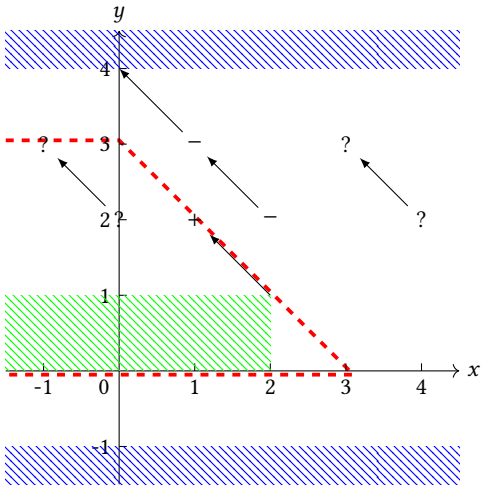


Figure 5: Learning result for example in Section 3.1

Example 4.6. Figure 5 shows the split result on the program in Example 3.1. In this figure, the green region represents a *true* (interval) example of $x \in (-\infty, 2], y \in [0, 1]$, and the two blue regions represent two *false* examples of $x \in (-\infty, +\infty), y \in (-\infty, -1]$ and $x \in (-\infty, +\infty), y \in [4, +\infty)$, respectively. Arrows in the figure represent implications. Initially, by applying the implication propagation, the examples $(1, 3)$ and $(2, 2)$ are classified as *false* (since $(0, 4)$ is *false*) and the example $(1, 2)$ is classified as *true* (since $(2, 1)$ is *true*). The learning algorithm then starts to split the examples. With three times of split (dashed red lines), all examples are classified. The final conjectured invariant is $y > -1 \wedge y \le 3 \wedge x + y \le 3$, with which the correctness of the program can be established.

5 EXPERIMENTS AND EVALUATION

We implemented a prototype[1] of our approach in the ICE learning framework [14]. This section reports experimental results of our approach on counterexample generalization techniques and interval-supported decision tree algorithm. The ICE implementation, combined with the corresponding decision tree as the learner (ICE-DT [15]), is the baseline for our experiments.

We use octagonal predicates (of the form $\pm x \pm y \le c$) as the templates for invariant synthesis. Inferred invariants are hence Boolean combinations of octagonal predicates instantiated from the templates. In some benchmarks, non-linear terms and other linear expressions are also used in templates. These additional templates are used in the ICE implementation. Our settings are the same as in the ICE implementation so as to perform a fair comparison. For boundary constraints and interval digging generalizations, the distances in $\mathcal{D}$ are $0, 1, 2, 3, 4, 5, 10, 20, 30, \ldots, 90, 100$. The constraint solver Z3 is used in our implementation [8].

Environment. All experiments are conducted on an Intel (R) Core (TM) computer with i7-9700 CPU (3.00 GHz) and 16 GB memory on Windows 10 platform.

[1]The archived link will be published after the double-blind review.

Benchmarks. 94 benchmarks are collected from two sources. ICE-DT provides 55 benchmarks; the remaining 39 benchmarks are from the loop-new, loop-lit, and loop-invgen categories in SV-COMP[2]. For the SV-COMP benchmarks, we remove the falsified programs to focus on loop invariant inference instead of bug finding. Moreover, we reduce large constants in 3 benchmarks to shorten verification time. In these 3 benchmarks, the verification time is explosive because the large constants lead to overmuch implication examples which our generalizations cannot speed up.

5.1 Experiment on Generalization Methods

Table 2 shows the experimental results. We only report benchmarks requiring at least 15 rounds in ICE-DT to save space. In the table, *ICE-DT* is the baseline [15]. *Elimination*, *Boundary*, and *Digging* correspond to generalizations by variable elimination, boundary constraint, and interval digging respectively. Each technique has four columns: the number of learning rounds R, the total time T, the time for constraint solver T_C, and the time for learner T_L. All time is in seconds. *Timeout* means the technique does not finish in a minute.

We summarize experiments in the bottom three lines of Table 2. All of our generalization techniques solve all and 2 additional benchmarks than ICE-DT. Also, our three techniques outperform ICE-DT significantly for the 92 benchmarks solved by the existing technique. The variable elimination generalization reduces about 35% in total round number and time. Boundary constraint and interval digging perform even better than variable elimination. For both generalization techniques, the total round numbers improve by 42% and the total time by 46%.

Further analyses on time for constraint solving and learning are instructive. Our generalization techniques require additional constraint queries with UNSAT core computation. Yet the time for constraint solving is reduced by 15% for variable elimination and by more than 26% for boundary constraint and interval digging. Moreover, our decision tree algorithm for interval counterexamples is more complex than standard algorithms. One would expect the new algorithm might be slower. Yet the total time for the new learner is improved by 45% for variable elimination and at least 56% for the other two techniques. This is because the new techniques need less rounds. Overhead in constraint solving and learning is compensated by the reduction in learning rounds.

To compare effectiveness among different generalization techniques, we further analyze the 89 non-trivial benchmarks verified by our techniques (Table 3). The 5 trivial benchmarks are verified by the trivial loop invariant $\top$ and hence ignored. In the table, *Bench* represents the ratio of benchmarks where generalization is effective. *Pos+Neg*, *Pos*, and *Neg* respectively represent the ratio of generalized non-implication, positive, and negative counterexamples. Generalization by variable elimination works for about half of the benchmarks. Boundary constraint and interval digging are effective for about 80% of the non-trivial benchmarks. In positive and negative counterexamples from constraint solvers, 23% are generalized by variable elimination and 47% by the other two techniques. This confirms our intuition. Removing variables from counterexamples is unlikely to generalize counterexamples. Interval generalization techniques such as boundary constraint or interval digging are more effective in our experiments.

Between positive and negative counterexamples, we find negative counterexamples are much easier to generalize. Recall that positive counterexamples are obtained from the entry location to loop head whereas negative counterexamples are from the loop head to assert location. Forward substitution used in positive counterexample generalization may fail and hence disable generalization.

Our implementation only asks Z3 to compute an UNSAT core. We have also compared the performance of *minimal* UNSAT cores in our implementation. Compared to generalization with UNSAT cores, we find round numbers are reduced but verification time is increased. Computing minimal UNSAT cores appears to have significant overhead but limited benefits.

5.2 Experiment on Learning Methods

Our generalization techniques improve the ICE framework by extracting more information from a counterexample. Standard decision tree inference algorithms might also benefit from generalized counterexamples and perform as well as our interval decision tree algorithm. It is therefore unclear whether our new decision tree algorithm is necessary. To answer this question, we design experiments to pass interval counterexamples to the baseline decision tree algorithm. More concretely, we use two counterexamples for each interval in an interval counterexample from the boundary constraint generalization technique. These counterexamples are then passed to the decision tree inference algorithm in ICE to mimic the interval counterexample. We would like to know if the baseline decision tree algorithm suffices to attain similar performance with similar information from generalized counterexamples. Figure 6(a) compares the number of rounds needed for the baseline and our interval decision tree algorithms. The total time is shown in Figure 6(b) and the time for learning is compared in Figure 6(c).

Round numbers in both inference algorithms are similar. Our new learning algorithm outperforms the baseline decision tree algorithm by about 7%. Some information is lost when an interval counterexample is simulated by several counterexamples. Yet the additional information is still useful compared to counterexamples without generalization. However, Figure 6(b) and 6(c) show that our new decision tree inference algorithm still outperforms the baseline algorithm in most benchmarks. On average, we find 5.7 counterexamples are needed for an interval counterexample. The baseline decision tree inference algorithm has to process many counterexamples and is hence less efficient. Our interval decision tree inference algorithm is essential in utilizing additional information from our generalization techniques.

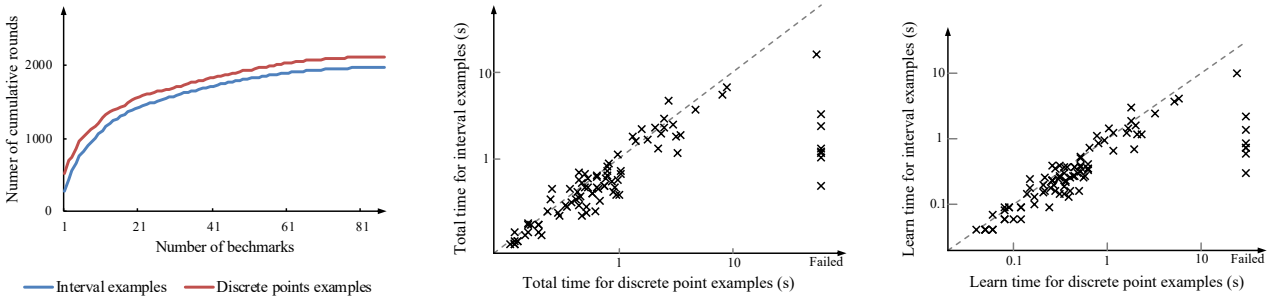
6 RELATED WORK

Invariant Synthesis. The traditional approaches for invariant synthesis are mainly focused on white-box techniques, which can be classified into the following categories: abstract interpretation-based [6, 7, 18, 25], interpolation-based [17, 21–24], counterexample guided abstraction refinement (CEGAR)-based [2, 5, 16, 39], logical abduction-based [3, 9] and so on.

[2]The SV-COMP benchmarks are available at https://github.com/sosy-lab/sv-benchmarks/tree/master/c.

Table 2: Experiment results on interval generalization methods

	ICE-DT				Elimination				Boundary				Digging			
Benchmarks	R	T	T_C	T_L	R	T	T_C	T_L	R	T	T_C	T_L	R	T	T_C	T_L
...	...				...				...				...			
dillig24	15	0.63	0.11	0.46	9	0.39	0.09	0.21	7	0.29	0.07	0.16	7	0.29	0.07	0.16
css2003	16	0.6	0.06	0.47	10	0.35	0.07	0.22	11	0.44	0.09	0.27	11	0.42	0.1	0.24
dillig28	17	0.7	0.09	0.52	17	0.62	0.11	0.39	16	0.61	0.11	0.36	16	0.61	0.11	0.37
sum4c	18	0.74	0.12	0.54	14	0.54	0.12	0.31	12	0.48	0.13	0.27	12	0.49	0.13	0.27
jm2006	19	0.74	0.08	0.57	9	0.33	0.06	0.21	15	0.56	0.11	0.34	15	0.6	0.12	0.35
matrixl2	19	0.87	0.2	0.57	13	0.63	0.19	0.33	16	0.72	0.26	0.36	16	0.74	0.27	0.37
sendmail-close	19	0.86	0.18	0.59	11	0.49	0.11	0.27	8	0.38	0.12	0.19	8	0.39	0.12	0.19
SpamAssassin	19	0.81	0.13	0.58	10	0.41	0.1	0.23	7	0.32	0.1	0.16	7	0.33	0.1	0.16
array	20	0.81	0.11	0.6	24	0.98	0.18	0.57	31	1.14	0.19	0.71	31	1.18	0.21	0.72
tacas	20	0.75	0.08	0.59	10	0.37	0.07	0.22	17	0.65	0.13	0.39	17	0.68	0.15	0.4
up	20	0.81	0.11	0.61	13	0.51	0.09	0.3	12	0.47	0.1	0.27	12	0.48	0.1	0.27
dillig17	21	0.83	0.1	0.62	16	0.65	0.13	0.37	21	0.81	0.13	0.49	21	0.81	0.14	0.49
dtuc	21	0.86	0.12	0.64	8	0.36	0.09	0.19	13	0.54	0.1	0.31	13	0.52	0.1	0.31
id_build	21	0.83	0.11	0.64	10	0.39	0.08	0.23	6	0.25	0.06	0.13	6	0.25	0.06	0.13
down	22	0.89	0.1	0.69	10	0.4	0.08	0.22	13	0.49	0.08	0.3	13	0.48	0.08	0.29
cegar2	22	0.84	0.08	0.65	28	1.1	0.17	0.66	16	0.59	0.1	0.36	16	0.61	0.11	0.37
gj2007b	23	0.91	0.1	0.7	11	0.44	0.08	0.25	12	0.46	0.08	0.27	12	0.46	0.09	0.27
nested-if3	23	0.92	0.11	0.69	11	0.44	0.08	0.26	11	0.42	0.08	0.25	11	0.42	0.08	0.25
arrayinv2	24	0.99	0.15	0.72	18	0.68	0.13	0.4	23	0.9	0.18	0.53	23	0.9	0.19	0.52
formula22	28	1.08	0.1	0.84	42	1.58	0.2	0.99	37	1.32	0.17	0.85	37	1.33	0.18	0.86
dillig05	30	1.18	0.13	0.92	20	0.82	0.11	0.47	14	0.57	0.13	0.33	14	0.56	0.13	0.32
half	30	1.24	0.17	0.94	41	1.88	0.38	1.12	29	1.22	0.26	0.72	29	1.29	0.26	0.75
nested6	30	1.36	0.26	0.95	17	0.77	0.19	0.41	15	0.68	0.19	0.35	15	0.7	0.2	0.36
matrixl2c	33	1.52	0.35	1.01	29	1.53	0.46	0.82	32	1.64	0.58	0.84	32	1.66	0.61	0.83
fragtest_simple	36	1.54	0.25	1.12	18	0.79	0.17	0.44	18	0.72	0.15	0.41	18	0.73	0.15	0.42
apache-escape	40	1.87	0.4	1.27	17	0.78	0.23	0.41	8	0.38	0.19	0.16	8	0.45	0.2	0.18
seq	45	2.19	0.39	1.56	31	1.61	0.36	0.93	33	1.69	0.42	0.96	33	1.76	0.47	0.97
sqrt	49	2.08	0.34	1.51	89	6.64	2.03	3.61	24	1.04	0.27	0.59	24	1.06	0.3	0.58
sum4	50	2.12	0.31	1.54	56	2.11	0.39	1.31	53	1.97	0.33	1.22	53	2.05	0.35	1.26
formula25	57	2.19	0.23	1.75	34	1.33	0.18	0.83	13	0.49	0.1	0.29	13	0.51	0.12	0.3
bhmr2007	58	2.6	0.39	1.93	57	2.91	0.47	1.81	26	1.17	0.3	0.67	26	1.2	0.34	0.67
gr2006	64	2.63	0.28	2.02	73	2.61	0.38	1.73	50	1.8	0.24	1.16	50	1.82	0.25	1.17
dillig12	70	3.22	0.42	2.37	89	4.66	0.8	3.03	61	2.54	0.44	1.56	61	2.57	0.45	1.58
cggmp	72	3.18	0.53	2.24	44	1.77	0.43	1.03	57	2.39	0.56	1.34	57	2.38	0.57	1.33
ddlm2013	76	3.18	0.45	2.4	41	2	0.33	1.28	36	1.78	0.4	1.09	36	1.84	0.47	1.08
dillig25	76	4	0.5	2.98	72	3.53	0.66	2.23	66	2.95	0.43	1.87	66	2.92	0.44	1.85
ex23	78	3.36	0.45	2.42	55	2.2	0.33	1.34	57	2.21	0.32	1.39	57	2.24	0.34	1.39
half2	80	3.79	0.59	2.75	48	2.25	0.44	1.33	44	1.9	0.36	1.14	44	1.94	0.39	1.15
cggmp2005	81	3.32	0.38	2.52	57	2.26	0.46	1.36	60	2.31	0.38	1.4	60	2.31	0.38	1.4
heapsort	82	3.94	0.8	2.71	32	1.73	0.39	1.03	27	1.3	0.41	0.66	27	1.35	0.44	0.67
inc	103	4.03	0.36	3.02	102	4	0.42	2.51	102	3.73	0.39	2.38	102	3.79	0.38	2.39
nested	120	9.02	1.47	6.71	86	5.95	1.38	3.6	68	3.32	0.57	2.12	68	3.36	0.61	2.13
formula27	137	5.49	0.56	4.35	69	3.32	0.38	2.27	29	1.18	0.21	0.74	24	0.99	0.2	0.6
arrayinv1	186	10.95	1.91	7.73	124	7.79	1.48	4.88	106	5.5	0.92	3.58	106	5.51	0.93	3.6
cggmp2005_b	186	8.78	1.27	6.07	159	6.28	1.01	3.88	168	6.67	0.98	4.04	168	6.77	0.98	4.05
gsv2008	205	8.49	1.06	6.49	11	0.42	0.08	0.26	7	0.28	0.07	0.15	7	0.29	0.07	0.16
large_const	411	21.56	4.18	14.02	23	1.12	0.31	0.59	43	2.32	0.67	1.21	43	2.42	0.73	1.23
gj2007		*Timeout*			206	10.25	1.78	5.89	281	16.66	3.01	9.61	281	16.67	3	9.63
string_concat		*Timeout*			118	4.98	0.75	3.11	112	4.67	0.62	2.91	112	4.71	0.65	2.87
# solved	92				94				94				94			
total	3191	147.44	27.2	106.24	2075	96.03	19.24	57.79	1835	77.39	15.96	45.69	1830	78.59	16.73	45.85
	-	-	-	-	35.0%	34.9%	15.2%	45.6%	42.5%	47.5%	29.7%	57.0%	42.7%	46.7%	26.3%	56.8%



(a) Cumulative rounds

(b) Total time comparison

(c) Learn time comparison

Figure 6: Experiment results on interval decision tree learning

Table 3: Generalization ability for different methods

	Elimination	Boundary	Digging
Bench	45/89 (50.6%)	71/89 (79.8%)	71/89 (79.8%)
Pos+Neg	268/1133 (23.7%)	384/811(47.3%)	380/807(47.1%)
Pos	37/207 (17.9%)	52/185 (28.1%)	52/185 (28.1%)
Neg	231/926 (24.9%)	332/626 (53.0%)	328/622 (52.7%)

Among black-box invariant learning techniques, Daikon [10] is an early but prominent one, which uses program states produced by dynamic test to learn conjunctive boolean invariants. The major shortcoming of Daikon is that the learned invariants may not be inductive, which was solved by Houdini [13]. With the help of constraint solvers, Houdini ensures the inductiveness of learned invariant. Garg et. al. [14] shows that merely learning from labeled examples for invariant synthesis is non-robust. They introduced the ICE learning framework for the implication counterexamples. A similar concept of implication examples is also proposed in [35]. However, because the pair of program state is unlabeled, [35] does not handle these examples directly.

There are a variety of techniques that can be used as the invariant learner, including random search [33], decision tree [11, 15, 19, 38], support vector machine [20, 36], PAC (Probably Approximately Correct) learning [35], reinforcement learning [37] and logic minimization [39], etc. Moreover, [34] uses the null space operation on matrix to learn the invariants in the form of algebraic equation. Garg et al. [15] extended the classical decision tree learning using implication examples. Their technique is called ICE-DT, which was further extended in [11] using Horn-ICE samples. Our interval-supported decision tree learning technique extends ICE-DT with interval examples. Compared to the previous decision trees [11, 15, 19, 38], our interval examples are more expressive. Given the same amount of examples, the learning result of our method is more likely to be more accurate.

Most of the learning approaches are format-restricted or template-based. For example, the decision tree learning is based on attribute templates which might need to be manually predefined. Many automatic feature inference techniques have emerged recently. PIE [26] uses a search-based program synthesis to achieve a strong expression ability of features. [4] mines existing codebase to provide features for static analysis. [38] utilizes machine learning to construct arbitrarily-linear features. These techniques are orthogonal to ours, and can be integrated into our approach. Actually, [38] show a vivid example where the learned features can further be delivered to the decision tree learner.

Counterexample Generalization. The purpose of generalization is to expand the amount of information to reduce iterations and improve performance. In white-box approaches, [21] generalizes the constraint conditions to construct the inductive invariants using the squeeze of interpolation. Another technique [1] also uses interpolation to find the generalized commonalities in positive or negative examples to help the correctness proof. Ivy [27] is an interactive system where the user can guide the counterexample generalization to help find the inductive invariants.

A similar idea in invariant learning is proposed in [20]. Except of the examples provided by the verification prover, [20] proposes the selective sampling by dynamic test execution to overcome the lack of sufficient examples. Another sampling technique in [12] suggests to find more "representative" examples by sampling from candidates with high priorities. Differing form these works, our generalization is based on results from constraint solvers rather than the dynamic testing. Note that the examples provided by the dynamic testing are relatively concrete.

7 CONCLUSION

We proposed a new invariant learning technique based on interval counterexamples. Three novel generalization techniques were devised to compute the interval counterexamples. The existing decision tree algorithm was improved to adapt interval counterexamples. We implemented a prototype of our approach in the ICE framework. Experimental results on 94 benchmarks show promising performance of our approach.

REFERENCES

[1] Muqsit Azeem, Kumar Madhukar, and R Venkatesh. 2018. Generalizing specific-instance interpolation proofs with SyGuS. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. 57–60.

[2] Thomas Ball and Sriram K Rajamani. 2001. The SLAM toolkit. In *International Conference on Computer Aided Verification*. Springer, 260–264.

[3] Cristiano Calcagno, Dino Distefano, and Viktor Vafeiadis. 2009. Bi-abductive resource invariant synthesis. In *Asian Symposium on Programming Languages and Systems*. Springer, 259–274.

[4] Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. 2017. Automatically generating features for learning program analysis heuristics for C-like languages. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 101.

[5] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)* 50, 5 (2003), 752–794.

[6] Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 269–282.

[7] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 84–96.

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

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

[10] Michael D Ernst, Adam Czeisler, William G Griswold, and David Notkin. 2000. Quickly detecting relevant program invariants. In *Proceedings of the 22nd international conference on Software engineering*. 449–458.

[11] P Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P Madhusudan. 2018. Horn-ICE learning for synthesizing invariants and contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[12] Grigory Fedyukovich, Samuel J Kaufman, and Rastislav Bodík. 2017. Sampling invariants from frequency distributions. In *2017 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 100–107.

[13] Cormac Flanagan and K Rustan M Leino. 2001. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe*. Springer, 500–517.

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

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

[16] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2003. Software verification with BLAST. In *International SPIN Workshop on Model Checking of Software*. Springer, 235–239.

[17] Ranjit Jhala and Kenneth L McMillan. 2006. A practical and complete approach to predicate refinement. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 459–473.

[18] Michael Karr. 1976. Affine relationships among variables of a program. *Acta informatica* 6, 2 (1976), 133–151.

[19] Siddharth Krishna, Christian Puhrsch, and Thomas Wies. 2015. Learning invariants using decision trees. *arXiv preprint arXiv:1501.04725* (2015).

[20] Jiaying Li, Jun Sun, Li Li, Quang Loc Le, and Shang-Wei Lin. 2017. Automatic loop-invariant generation and refinement through selective sampling. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 782–792.

[21] 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 *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 793–803.

[22] Kenneth L McMillan. 2003. Interpolation and SAT-based model checking. In *International Conference on Computer Aided Verification*. Springer, 1–13.

[23] Kenneth L McMillan. 2006. Lazy abstraction with interpolants. In *International Conference on Computer Aided Verification*. Springer, 123–136.

[24] Kenneth L McMillan. 2010. Lazy annotation for program testing and verification. In *International Conference on Computer Aided Verification*. Springer, 104–118.

[25] Antoine Miné. 2006. The octagon abstract domain. *Higher-order and symbolic computation* 19, 1 (2006), 31–100.

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

[27] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 614–630.

[28] J. Ross Quinlan. 1986. Induction of decision trees. *Machine learning* 1, 1 (1986), 81–106.

[29] J. Ross Quinlan. 1993. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. http://portal.acm.org/citation.cfm?id=152181

[30] Lior Rokach and Oded Z Maimon. 2008. *Data mining with decision trees: theory and applications*. Vol. 69. World scientific.

[31] S Rasoul Safavian and David Landgrebe. 1991. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics* 21, 3 (1991), 660–674.

[32] Claude E Shannon. 1948. A mathematical theory of communication. *Bell system technical journal* 27, 3 (1948), 379–423.

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

[34] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V Nori. 2013. A data driven approach for algebraic loop invariants. In *European Symposium on Programming*. Springer, 574–592.

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

[36] Rahul Sharma, Aditya V Nori, and Alex Aiken. 2012. Interpolants as classifiers. In *International Conference on Computer Aided Verification*. Springer, 71–87.

[37] 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*. 7751–7762.

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

[39] He Zhu, Aditya V Nori, and Suresh Jagannathan. 2015. Learning refinement types. *ACM SIGPLAN Notices* 50, 9 (2015), 400–411.