Learning Disjunctive Invariants based on Loop Structures

Anonymous Author(s)

ABSTRACT

Loop-invariants are crucial in program analysis and verification. Many approaches based on machine learning have been proposed to generate loop-invariant automatically. Existing learning-based approaches, however, have limited capability in generating disjunctive loop invariants. In this work, we present a novel technique to learn disjunctive loop-invariants based on the loop structure. We partition program states according to not only whether they satisfy the loop-invariant to be generated but also which part in the loop they visit. Classification techniques are then employed to construct a predicate for each partition, separating program states which satisfy the loop-invariant from those do not. Subsequently, a disjunction of these predicates and the path condition for each partition is generated as a candidate loop invariant. Afterwards, we check whether the program can be verified with the candidate invariant. If not, counterexamples are generated to help refine our learning. Furthermore, in order to learn simpler invariants (invariant with fewer disjunctive clauses), we further propose a technique to unwind the program structure adaptively. The evaluation results show that our approach can effectively learn disjunctive loop invariants.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**;

KEYWORDS

Disjunctive loop-invariant, program verification, machine learning, loop structure, graph cut, program abstraction

ACM Reference format:
Anonymous Author(s). 2018. Learning Disjunctive Invariants based on Loop Structures. In *Proceedings of 40th International Conference on Software Engineering, Gothenburg, Sweden, May 27 - June 3 (ICSE'2018),* 11 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In order to reason about the correctness of programs, Hoare Logic was introduced in [32], where loop-invariants are central to verify loop-containing programs against its specification. If an invariant is provided for each loop in the program, the problem of program verification can be solved through logical deduction using automated constraint solvers. In addition, loop-invariants are useful in compiler optimization, program understanding, etc. Generating loop-invariants automatically, however, is highly challenging, which has since become a key problem in program verification.

Many approaches have been proposed for loop-invariant generation, e.g., template-based approaches [10, 17, 29] and approaches based on abstract interpretation [18, 25, 43]. However, generating disjunctive loop-invariants, which are required for proving the correctness of certain programs, is still considered as a highly non-trivial problem. Among the existing approaches, template-based approaches documented in [10, 17, 29] may generate disjunctive invariants if the user provides disjunctive templates. However, knowing what disjunctive templates are relevant is often challenging. Approaches based on abstract interpretation [6, 9, 19, 40] have limited support for disjunctive invariants due to the complexity in maintaining such expressive abstract domains.

In recent years, a number of learning-based approaches have been proposed [3, 23, 34, 46–48], some of which are capable of learning disjunctive loop-invariants. These approaches dynamically execute the program and gather program states at run-time. Based on the gathered program states and their labeling (i.e., whether they should satisfy the loop invariant to be generated or not), a candidate loop-invariant is generated using machine learning techniques. Sharma *et. al.* [48] proposed the *SVM-I* algorithm, which is based on the classical support vector machines (*SVMs*) [12, 41] algorithm, to learn conjunctive invariants. The algorithm can be applied to learn one particular form of disjunctive loop-invariants if we reverse the labels on the training sets and take the negation of the learned predicate as the candidate loop-invariant. In [24], Garg *et. al.* proposed to learn loop-invariants in the form of an arbitrary boolean combination of a set of predicate templates through decision trees [42]. Like template-based approaches, the predicate templates must be pre-defined. Thus, only simple templates like $\pm x \pm y \leq c$ where $|c| \leq m$ are used in their work. In [47], the authors proposed to apply PAC learning techniques for invariant generation. It has been demonstrated that their approach may learn invariants in the form of arbitrary boolean combinations of a given set of propositions. Similarly, the propositions must be pre-defined. In addition, Sharma *et. al.* [45] proposed to decompose *multi-phase* loops into several single-phase loops with the 'splitter predicates'. Through this transformation, a loop with disjunctive invariants can be converted into several loops with non-disjunctive loop-invariants. Their approach however only applies to a limited class of loops.

In this work, we propose a new approach to generate disjunctive loop-invariants. Central to our work is the observation that a disjunctive loop invariant is necessary often due to branching statements in the loop. That is, often different invariants might be satisfied for different branches in the loop. We thus design our learning algorithm to utilize the knowledge of the structure of the loop. Our approach is based on existing learning-based approaches [36, 48]. We collect at run-time the program states when the loop head is reached. Different from existing approaches, we partition the program states (if necessary) according to which branches

ICSE'2018, Gothenburg, Sweden

DOI: 10.1145/nnnnnnn.nnnnnnn

each program state visits in the loop. A valid partitioning is defined based on a *cut* on the loop structure. For each partition, a hypothesis is learned through classification algorithms. Furthermore, each partition is associated with a path condition which defines the condition required to visit that partition. Afterwards, a candidate loop-invariant is constructed as the disjunction of the hypothesis conjuncted with the corresponding path condition.

Since the number of clauses in the resulting invariant depends on the number of partitions, it is desirable to have a cut which results in a small number of partitions. Choosing an improper cut may lead to partition explosion. Therefore, we design an algorithm to heuristically find a good cut by adaptively traversing the loop structure in a top-down manner. It assumes loop invariants can be synthesized in a way similar to how programs are developed.

In summary, our work makes the following contributions.

- Firstly, we present an approach to partition program states based on loop structure and design an algorithm to learn disjunctive invariants based on the partitions. We show that multiple existing approaches [36, 45, 48] can be regarded as special cases of our approach.
- Secondly, we design a heuristic algorithm to search for a good program partitioning which would lead to a disjunctive loop-invariant with fewer clauses by exploring the structure of the loop adaptively.
- Lastly, we implement our approach as a prototype tool and evaluate its performance with a comprehensive set of benchmarks. The results show that our approach is more effective in learning disjunctive loop-invariants compared with existing approaches.

The remainders of the paper are organized as follows. Section 2 defines the loop-invariant generation problem. Section 3 presents our approach in details. Section 4 presents the cut search algorithm. Section 5 evaluates our approach. Section 6 reviews related work and Section 7 concludes.

2 INVARIANT LEARNING

In this section, we first define the loop-invariant generation program. Let $V = \{x_1, x_2, \cdots, x_n\}$ be a finite set of program variables. For simplicity, we assume they are of type boolean, integer, float or double. A valuation of the program variables, $s = \{x_1 \mapsto v_1, x_2 \mapsto v_2, \cdots, x_n \mapsto v_n\}$ where $v_1, v_2, \cdots, v_n$ are constants, is called a *program state*. For simplicity, it is written as $s = (v_1, v_2, \cdots, v_n)$ when the set of variables are clear from the context. Let ϕ be a predicate constituted by variables in V. ϕ is viewed as the set of valuations of V such that ϕ evaluates to true. We thus write $s \in \phi$ to denote that ϕ is evaluated to *true* given s. Otherwise, we write $s \notin \phi$. For example, if a program contains two variables x and y, $V = \{x, y\}$ and a program state is of the form $s = (a, b)$ where a and b are values of x and y. Let s be $(0, 1)$; ϕ_1 be $x + y > 0$; and ϕ_2 be $x > 0$. We have $s \in \phi_1$ and $s \notin \phi_2$.

2.1 Loop Invariant

For simplicity, we assume that a program and its correctness specification are given in the form of a Hoare triple:

$$\{Pre\}\ while(Cond)\ \{Body\}\ \{Post\}$$

where Pre is the precondition; $Post$ is the postcondition; $Cond$ is the loop condition and $Body$ is the loop body. Pre, $Cond$ and $Post$ as predicates constituted by variables in V. $Body$ is an imperative program that updates variables in V. The Hoare triple is valid, if for all the program states $s \in Pre$, when we execute the program with initial program state s, after the loop terminates, the resulting program state $s' \in Post$. Equivalently, we say that the program is correct with respect to the precondition and postcondition. For simplicity, we assume that the given program is deterministic with respect to the postcondition. That is, given any program state s, if we execute the program with initial program state s, after the loop terminates, the resulting program state either always satisfies the postcondition or always fails the postcondition.

The goal of loop verification is to either prove or disprove the Hoare triple. To prove it, we would like to find a loop-invariant Inv which satisfies three conditions [32].

$$Pre \Rightarrow Inv \tag{1}$$

$$\{Inv \wedge Cond\}\ Body\ \{Inv\} \tag{2}$$

$$Inv \wedge \neg Cond \Rightarrow Post \tag{3}$$

where Inv is a loop-invariant. Note that if the Hoare triple is valid, there must exist a loop-invariant Inv satisfying the three conditions [32]. To disprove a Hoare triple, we would like to find a variable valuation s such that $s \in Pre$ and executing the loop until it terminates results in a valuation s' such that $s' \notin Post$. For simplicity, we assume the loop always terminates and refer the readers to [4, 15] for research on proving loop termination.

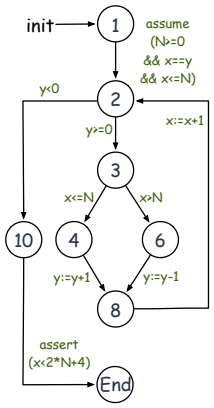
Example 2.1. An example Hoare triple is shown in Figure 1a, where the `assume` statement captures the precondition and the `assert` statement captures the postcondition. The variables of the program are x, y and N, all of which are integers. Note that we interpret integers as mathematical integers (i.e., they do not overflow). Given a Hoare triple, we can always test its validity through testing. If we test the program from the program state $s = (0, 0, 0)$, the precondition is satisfied and the program state $s = (3, -1, 0)$ is reached after the loop which satisfies the postcondition. We can prove the program with $Inv \equiv (y < 0 \wedge x < 2 * N + 4) \vee (y \geq 0 \wedge x \leq N \wedge x \geq y) \vee (y \geq 0 \wedge x > N \wedge x + y \leq 2 * N + 2)$.

2.2 Guess and Check

Following [36, 48], we adopt a guess-and-check approach to form loop-invariant and check its validity repeatedly. The overall approach is shown in Algorithm 1. We start with randomly generating a set of valuations of V, denoted as SP, at line 1 (a.k.a. random sampling). Random sampling provides us an initial set of samples to learn the first candidate loop-invariant. Next, for any valuation s in SP, we execute the program starting with variable valuation s and record the valuation of V after each iteration of the loop. We write $s \Rightarrow s'$ to denote that if we start with valuation s, we obtain s' after some number of iterations. At line 3 of Algorithm 1, we add all such valuations s' into SP. Next, we categorize SP into four disjoint sets: CE, $Positive$, $Negative$ and NP. Intuitively, CE contains program states which can be used to disprove the Hoare triple; $Positive$ contains those program states which satisfy any loop invariant which proves the Hoare triple; $Negative$ contains those program

```
1   assume(N ≥ 0
           && x = y
           && x ≤ N);
2   while(y ≥ 0) {
3     if (x ≤ N) {
4       y := y + 1;
5     } else {
6       y := y − 1;
7     }
8     x := x + 1;
9   }
10  assert(x < 2 ∗ N + 4);
```

(a) A Hoare triple



(b) The transition system

Figure 1: An illustrative Example

states which must not satisfy any loop invariant which proves the Hoare triple; and *NP* contains the rest. Formally,

$$CE(SP) = \{s \in SP | s \in Pre \wedge \exists s'. \; s \Rightarrow s' \wedge s' \notin Cond \wedge s' \notin Post\}$$

A valuation s in $CE(SP)$ starts from a valuation s_0 which satisfies Pre and becomes a valuation s' which fails $Post$ when the loop terminates. If $CE(SP)$ is non-empty, the Hoare triple is disproved.

$$Positive(SP) = \{s | \exists s_0 \in SP. \; \exists s'. \; s_0 \Rightarrow s' \Rightarrow s \Rightarrow s_n \wedge s' \models Pre \wedge s_n \not\models Cond \wedge s_n \models Post\}$$

$Positive(SP)$ contains a valuation s if there exists a valuation s_0 in SP which satisfies Pre and becomes s after zero or more iterations. Furthermore, s subsequently becomes s', which satisfies $Post$ when the loop terminates.

$$Negative(SP) = \{s \in SP | \exists s_0, s'. s_0 \notin Pre \wedge s_0 \Rightarrow s \Rightarrow s' \wedge s' \notin Cond \wedge s' \notin Post\}$$

$Negative(SP)$ is a valuation s which starts from a valuation s_0 violating Pre and becomes a valuation s' which violates $Post$ when the loop terminates.

$$NP(SP) = SP - CE(SP) - Positive(SP) - Negative(SP)$$

$NP(SP)$ contains the rest of the samples.

Proposition 2.2. *[36] If the Hoare triple is valid, there exists a loop invariant Inv satisfying condition (1), (2) and (3) such that* $Positive(SP) \subseteq Inv$ *and* $Negative(SP) \cap Inv = \emptyset$.

A program state in $NP(SP)$ may or may not satisfy an invariant Inv which satisfies condition (1), (2) and (3).

After obtaining the samples, method $learn(SP)$ at line 4 in Algorithm 1 is invoked to generate a candidate invariant ϕ. We leave the details on how a candidate loop-invariant are generated in later subsections, which is the contribution of this work. Once a candidate is identified, we move on to improve it through selective sampling. The basic idea is to generate additional test cases close to the boundary between program states which satisfy ϕ and those which violate ϕ so as to refine ϕ. Since it is not the contribution

Algorithm 1: Algorithm *GuessAndCheck*()

```
1  Let SP be a set of randomly generated valuations of V;
2  while not time out do
3      Add all valuations s' s.t. s ⇒ s' for s ∈ SP into SP;
4      Call learn(SP, P) to generate a loop-invariant φ;
5      Improve φ through selective sampling;
6      if φ is null then
7          Report "failed" and break;
8      if φ satisfies (1), (2) or (3) then
9          Report "proved" and break;
10     Add program states violating (1), (2) or (3) into SP;
```

of this work, we refer the readers to [36] for details. At line 8, we check whether ϕ satisfies the three conditions. In particular, we check whether any of the following constraints is satisfiable or not using an SMT solver [7, 20].

$$Pre \wedge \neg\phi \tag{4}$$

$$sp(\phi \wedge Cond, Body) \wedge \neg\phi \tag{5}$$

$$\phi \wedge \neg Cond \wedge \neg Post \tag{6}$$

where $sp(\phi \wedge Cond, Body)$ is the strongest postcondition obtained by symbolically executing program $Body$ starting from precondition $\phi \wedge Cond$ [21]. If all the three constraints are unsatisfiable, we successfully prove the Hoare triple with the loop invariant ϕ. If any of the constraints is satisfiable, a model in the form of a variable valuation is generated, which is then added to SP as a new sample. Afterwards, we restart from line 2, i.e., we execute the program with the counterexample valuations, collect and add the variable valuations after each iteration of the loop to the four categories accordingly, move on to learning and so on.

Example 2.3. For the program in Figure 1, assume the initial variable valuation set is $SP = \{(0,0,0),(0,1,0)\}$. After executing the program with these initial valuations, we can collect the valuations after one or more iterations of the loop. These valuations are then grouped into the four sets as described above. In particular, $Positive(SP) = \{(0,0,0),(1,1,0),(2,0,0),(3,-1,0)\}$, and $Negative(SP) = \{(0,1,0),(1,2,0),(2,1,0),(3,0,0),(4,-1,0)\}$ and the other two sets are empty. We then make a guess of the invariant Inv by learning a classifier such that $Positive(SP) \subseteq Inv$ and $Negative(SP) \cap Inv = \emptyset$ are satisfied. Then we can check whether we are able to verify the program with Inv.

2.3 Classification

Like existing learning-based approaches for loop-invariant generation [36, 48], we employ classification techniques to generate candidate loop invariants, i.e., to implement function $learn(SP, \mathcal{P})$ in Algorithm 1. In the following, we briefly introduce how classification algorithms work.

In the simplest setting, it is assumed that there is a training set containing two sets of labeled samples, i.e., set P contains samples which are labeled as positive and set N contains samples which are labeled as negative. The goal is to learn a function f which accurately predicts the labels of samples arising in the future. In the

setting of generating loop-invariants, classification techniques are adopted to generate classifiers for $Positive(SP)$ and $Negative(SP)$ as candidate loop-invariants. Note that since $NP(SP)$ may or may not satisfy the loop-invariant to be generated. As suggested in [36], we can additionally generate classifiers separating $Positive(SP)$ from $Negative(SP)$ and $NP(SP)$ or separating $Negative(SP)$ from $Positive(SP)$ and $NP(SP)$ as candidates. For simplicity, we focus on generating classifiers for $Positive(SP)$ and $Negative(SP)$ in this work.

There are many existing classification algorithms. For instance, the decision tree algorithm [42] learns classifiers in the form of boolean combinations of a given set of propositions (a.k.a. features). It constructs simple classifiers by iteratively discovering propositions which maximize information gain. A commonly applied supervised classification algorithm is *SVM* [12]. *SVM* is a supervised machine learning algorithm for classification and regression analysis [12]. In general, the binary classification functionality of *SVM* works as follows. Given P and N, *SVM* generates a perfect classifier to separate them if there is any. Note that in our setting, any classification error is intolerable and thus the classification algorithms must be tuned to generate perfect classifiers. Formally, a perfect classifier ϕ for two sets P and N is a predicate such that $s \in \phi$ for all $s \in P$ and $s \notin \phi$ for all $s \in N$. *SVM* by default learns classifiers in the form of a linear inequality, i.e., a half space in the form of $c_1x_1 + c_2x_2 + \cdots \geq k$ where x_i are variables in V and c_i are constant coefficients. The *SVM-I* algorithm proposed in [48] extends *SVM* to learn classifiers in the form of a conjunction of multiple linear inequalities. The algorithm is further extended to learn conjunctions of polynomial inequalities in [36].

As shown in [36], *SVM* and *SVM-I* can be very powerful in learning conjunctive invariants. However, the program in Figure 1a can not be handled with these algorithms, as the required invariant contains both conjunction and disjunction.

3 LEARNING BASED ON STRUCTURE

Next, we present the details of our invariant learning approach based on program structures. In the following, we fix a Hoare triple $\{Pre\}\ while(Cond)\ \{Body\}\ \{Post\}$.

3.1 Program Structure

Without loss of generality, we assume that the program can be modeled as a transition system $\mathcal{P} = (S, init, T, L)$. S is a finite set of states, corresponding to the control locations in the program. $init \in S$ is a unique entry point (i.e., the start of the loop). $T \subseteq S \times S$ is a set of edges which captures the control flow, i.e., an edge from state s to s' means that the control flow can transit from the control location represented by s to that presented by s'. Lastly, L is a labeling function which labels each transition with a guarded command $[g]f$ where g is a guard condition and f is a function updating variable valuation. Note that g is used to model branching conditions whereas f is used to model variable-updating statements like assignments. For instance, Figure 1b shows the transition system (a.k.a. control flow graph) of the program in Figure 1a.

A program path is a sequence of connected transitions $\pi = \langle (c_1, c_2), (c_2, c_3), \cdots, (c_k, c_{k+1}) \rangle$ such that $c_1 = init$ and $(c_i, c_{i+1}) \in T$ for all i. We write $paths(\mathcal{P})$ to denote all finite paths of $\mathcal{P}$. Let $[g_i]f_i$ be $L((c_i, c_{i+1}))$ for all i. We can construct the corresponding path condition as: $PC(\pi) = \exists v_2, \cdots, v_{k+1}.\ g_1 \wedge (v_2 = f_1(v_1)) \wedge g_2 \wedge \cdots \wedge g_k \wedge (v_{k+1} = f_k(v_k))$. A test execution of $\mathcal{P}$ from a program state v_0 is a sequence $\pi = \langle (v_0, s_0), t_0, (v_1, s_1), t_1, \cdots, (v_k, s_k), t_k, \cdots \rangle$ where v_i is a valuation of V, $s_i \in S$, $L(t_i) = [g_i]f_i$ and $v_i \in g_i$, and $v_{i+1} = f_i(v_i)$ for all i, and $c_0 = init$.

3.2 Program States Partitioning

As discussed above, existing learning-based approaches generate candidate loop-invariants by learning a classifier separating $Positive(SP)$ and $Negative(SP)$. For Hoare triples which require disjunctive loop-invariants to prove, we may find that there is no classifier which can completely separate $Positive(SP)$ and $Negative(SP)$.

In the following, we explain why partitioning the program states in $Positive(SP)$ and $Negative(SP)$ might be useful to learn disjunctive loop invariants. Assume the actual loop-invariant is in the disjunctive normal form of $\phi_1 \vee \phi_2 \vee \cdots \vee \phi_n$ where ϕ_i is a conjunctive clause for all i. If we partition the program states in $Positive(SP)$ and $Negative(SP)$ into n partitions such that P_i contains all those program states in $Positive(SP)$ which satisfy ϕ_i and N_i contains all those program states in $Negative(SP)$ which satisfy ϕ_i, there must be a conjunctive classifier which perfectly separate P_i and N_i, which could be potentially learned using the *SVM-I* (or *SVM*) algorithm. The issue is of course how we partition the program states without knowing ϕ_i. Different from the data driven approach, i.e. ICE with decision trees, which can be viewed as partitioning program states based on predicates, we propose to use the structure information to partition program states. In order to partition program states based on the program structure, we define the notion of cuts in the following.

Definition 3.1. A cut of a program $\mathcal{P} = (S, init, T, L)$ is a set of transitions $C \subseteq T$ such that at least one transition in C is visited by any test execution of the program.

Given a cut C and a set of program states S, we can partition the program states in S according to which transition in C is visited *first*. That is, the result of the partition is a set of set of program states, i.e., each transition t in C is associated with a subset of program states in S such that for each s in S, the test execution from s visits t before any other transitions in C. We write $partition_t(S, C)$ where $t \in C$ to denote the partition which is associated with transition t.

Example 3.2. For the program shown in Figure 1a, if the cut is $\{(1, 2)\}$, there is only one partition, which contains all the program states. If the cut is $\{(2, 3), (2, 10)\}$, the program states are partitioned into two, one containing those entering the loop and the other containing those skipping the loop. If the cut is $\{(2, 10), (3, 4), (3, 6)\}$, the program states are partitioned into three: one containing those skipping the loop, one containing those satisfying the condition at line 3 (in the current iteration) and one containing the rest. $\{(3, 4), (3, 6)\}$ is not a valid cut according to our definition.

The following states that given a cut, we can uniquely partition every program state.

Proposition 3.3. *Let C be any cut and SP be any set of program states.*

$$\cup_{t \in C} partition_t(SP, C) = SP$$

Algorithm 2: Algorithm $generate(SP, C)$

```
1  let invariant I := false;
2  let fts be ∅;
3  fts for each transition t in C do
4      Obtain PC_t;
5      Let P = partition_t(Positive(SP), C);
6      Let N = partition_t(Negative(SP), C);
7      Let φ_t be SVM-I(P, N);
8      if φ_t ≠ NULL then
9          I = I ∨ (PC_t ∧ φ_t);
10     else
11         Add t into fts;
12 return (I, fts);
```

and

$$partition_c(SP, C) \cap partition_d(SP, C) = \emptyset$$

for all $\{c, d\} \subseteq C$. □

We say that two cuts are equivalent if and only if for any set of program states, the cuts result in the same partitioning of the program states. For instance, for the program shown in Figure 1a, the two cuts $\{(2,10),(2,3)\}$ and $\{(2,10),(8,2)\}$ are equivalent.

Next, we associate each transition t in cut C with a path condition PC_t. Intuitively, PC_t is the path condition which must be satisfied by any test execution which visits t before any other transition in C. Formally, PC_t is defined as follows: $\bigvee_{\pi \in path(\mathcal{P},t)} PC(\pi)$ where $path(\mathcal{P}, t)$ is the smallest set of finite paths of $\mathcal{P}$ such that: for all $\langle t_1, t_2, \cdots, t_n \rangle \in path(\mathcal{P}, c)$, $t_n = t$ and $t_i \notin C$ for all $1 \leq i < n$.

Lemma 3.4. *Given any cut* C, $\vee_{t \in C}\{PC_t\} = true$.

Proof. We prove this by contradiction. Assume that $\vee_{t \in C}\{PC_t\} = true$ is not true. There must exist a program state s which does not satisfy any of the PC_t, which implies that s does not visit any of the transitions set in C and thus contradicts the definition of cuts. □

The following theorem forms the basis of our approach on partitioning and learning.

Theorem 3.5. *Given any cut* C*, the Hoare triple is valid if and only if there exists some predicate* ϕ_t *for all* $t \in C$ *such that* $\bigvee_{t \in C} PC_t \wedge \phi_t$ *is a loop-invariant satisfying the three conditions.*

Proof. **if**: It is trivially true. **only-if**: If the Hoare triple is valid, there exists an invariant inv which satisfies rule 1, 2 and 3. Let ϕ_t be inv for all t. We have

$$\begin{aligned} \vee_{t \in C}\{PC_t \wedge \phi_t\} &\equiv \vee_{t \in C}\{PC_t \wedge inv\} \\ &\equiv \vee_{t \in C}\{PC_t\} \wedge inv \\ &\equiv inv \qquad \text{by Lemma 3.4} \end{aligned}$$

□

Based on the above theorem, in order to learn disjunctive loop invariant, one way is to learn ϕ_t for each t in a given cut. The following establishes that each ϕ_t can be learned through learning classifiers separating program states in $Positive(SP)$ and $Negative(SP)$ in the corresponding partition.

Proposition 3.6. *For any transition* t *in a cut* C,

$$partition_t(Positive(SP), C) \subseteq \phi_t$$

and

$$partition_t(Negative(SP), C) \cap \phi_t = \emptyset$$

As a consequence, for each transition t in C, a classifier ϕ_t separating $partition_t(Positive(SP), C)$ and $partition_t(Negative(SP), C)$ perfectly can be generated. Afterwards, we can construct a candidate loop-invariant through $\vee_{t \in C}\{PC_t \wedge \phi_t\}$. Intuitively, we are more likely to generate ϕ_t than generating a classifier separating $Positive(SP)$ and $Negative(SP)$ since there are fewer program states in $partition_t(Positive(SP), C)$ and $partition_t(Negative(SP), C)$.

Algorithm 2 shows the details on generating a loop invariant. It takes the the set of program states SP and a cut C as inputs. There are two variables: I is the placeholder for the invariant and the set fts contains all and only transition t such that we cannot generate the classifier ϕ_t. Note that $fts \subset C$. I is initialized as $false$, and fts is initially empty. In the loop from line 3 to 11, the algorithm enumerates all the transitions in C. For each transition t, the path condition PC_t is calculated at line 4. For programs with no nested loops, the set of all paths reaching a transition t, written as $path(\mathcal{P}, t)$, is finite and thus we construct PC_t as the disjunction of path condition of all paths in $path(\mathcal{P}, t)$. For programs with nested loops, the set $path(\mathcal{P}, t)$ is unbounded and thus we cannot enumerate them. However, a close look at the proof of Theorem 5 show that we can replace PC_t with an over-approximation and the proof still stands. Thus, in this work, for programs with nested loops, we simply set PC_t to be true.

At line 7, the *SVM-I* (or *SVM* for simplicity) algorithm is employed for learning a classifier in the form of a conjunctive inequality (or a linear inequality). If the classification succeeds, we update I with a new disjunctive clause, i.e., $PC_t \wedge \phi_t$. Otherwise, we add t into fts (so that a different *cut* will be used as we explain in the next subsection). There are two outputs. One is that fts is an empty set after the loop is finished, in which case an invariant candidate has been successfully generated. The other is that fts is not empty, in which case we identify a different cut and try again.

We note that for the partition $partition_t$ which skips the loop body (e.g., the partition corresponding to the transition $(2, 10)$ for the example in Figure 1b), ϕ_t can be obtained directly from the postcondition $Post$, as all the variable valuations in this partition should satisfy $Post$, or else a counter-example can be generated to disprove the triple. Moreover, in our setting, $\phi_t = Post$ is also adequate for the proving the three conditions. Nevertheless, we do not distinguish such partition with the others in the following.

Example 3.7. For the program in Figure 1, assuming we are given a cut $C = \{(2,10),(3,4),(3,6)\}$, we partition program states according to which path a program state takes, i.e., either skipping the loop, taking the if-branch or taking the else-branch. Based on C, we partition the states in Example 2.3 and obtain the path conditions and partition predicates as shown in Table 1 where P_t is short for $partition_t(Positive(SP), C)$ and N_t is short for $partition_t(Negative(SP), C)$. The predicate $\phi_{(2,10)}$ equals to $Post$. Other predicates ϕ_t can be generated through SVM algorithm (if more program states are provided). With these path conditions and partition predicates, the loop invariant shown in Example 2.3 can be composed.

t	(2,10)	(3,4)	(3,6)
PC_t	$y < 0$	$y \geq 0 \wedge x \leq N$	$y \geq 0 \wedge x > N$
P_t	(3,-1,0)	(0,0,0)	(1,1,0),(2,0,0)
N_t	(4,-1,0)	(0,1,0)	(1,2,0),(2,1,0),(3,0,0)
ϕ_t	$x < 2 * N + 4$	$x \geq y$	$x + y \leq 2 * N + 2$

Table 1: Example partition

4 SEARCHING FOR THE CUT

In our approach, a cut with too many transitions would result in a disjunctive invariant with many clauses, whereas a cut with too few transition may result in no learned invariant at all if if some partitions are inseparable. Therefore, finding a good cut is a key problem in our approach.

Existing approaches in [36, 48] can be viewed as special cases of our approach where two particular kinds of cuts are adopted. Previous learning-and-guess approaches which do not explore the loop structure can be viewed as using a cut which results in one partition only. In such a case, we ignore the program structure and all program states SP are contained in one partition. For the example in Figure 1a, one such cut is $\{(1, 2)\}$. The benefit of having one partition only is that if a candidate loop-invariant is generated, it is simple (i.e., having only one clause). However, if $Positive(SP)$ and $Negative(SP)$ are inseparable (i.e., a disjunctive loop-invariant is required), no candidate loop-invariant is identified.

In [36], the authors briefly discussed the idea of partitioning program states according to the program path a program state visits in the loop. This approach can be viewed as using a cut which contains one transition for each path in the loop. For the example in Figure 1a, one such cut is $\{(2, 10), (3, 4), (3, 6)\}$. While having such a cut may allow the approach in [36] to learn some disjunctive loop invariants, it has two problems. One is that it dose not apply to programs with nested loops since the number of paths in the loop in unbounded due to the inner loop. The other is that there might be many paths in the loop which leads to many partitions and a complicated disjunctive loop-invariant with many clauses (which may require many samples to learn). For instance, if there are n number of conditional statements in the program, there might be 2^n paths in the worst case. Having a overly complicated loop-invariant is not ideal since it may be challenging to verify. Ideally, we would like to learn loop-invariants which are simple and yet satisfy the three conditions. For instance, the invariant learned for the program in Figure 1a using such a cut is more complicated than needed. We will show how to learn a 'better' invariant in the following.

In order to identify 'better' cuts, we first define a partial order on cuts.

Definition 4.1. A cut C is better than a cut C', written as $C' \leq C$, if $\#C \leq \#C'$ where $\#C$ is the number of transitions in C and both C and C' result in a candidate loop-invariant, or C results in a candidate loop-invariant and C' does not.

Our aim is thus to find a 'best' cut, i.e., one such that there does not exist a better cut. Intuitively, by enumerating all the possible cuts associated with a given transition system, we can find the 'best' one by comparing each of them according to the above definition.

Algorithm 3: Algorithm $learn(SP, \mathcal{P})$

```
1  Let C be a set containing all outgoing transitions from init;
2  Let expanded and visited both be ∅;
3  while true do
4      if C ∈ visited then
5          return null;
6      Add C into visited;
7      Let (I, fts) := generate(SP, C);
8      if fts = ∅ then
9          return I;
10     for each transition (s, s') in fts do
11         if (s, s') is in expanded then
12             continue;
13         expanded := expanded ∪ {(s, s')};
14         if s' has no outgoing transition then
15             return null;
16         Substitute (s, s') with outgoing transitions of s' in C;
```

Such algorithm is impractical as the number of possible cuts are exponential in the number of transitions. Therefore, we propose a heuristic algorithm to find a good cut, by incrementally exploring the loop structure in a top-down manner.

Intuitively, our approach is based on the idea that the structure of a loop (and a program in general) is a reflection of the programming model in the developer's mind. When we read a program, we often do a top-down manner (i.e., read from the beginning of a program and explore the inner structure of a block of code only if we cannot develop an understanding of the block without its inner details). We believe that loop-invariant generation should follow a similar process. For instance, given the program shown in Fig. 1a where there is a branch in the loop. If we can learn an invariant regarding the loop as a black-box, we would prefer not to explore the structure of the loop. Only if we could not learn regarding the loop as a black-box, we explore the loop structure. Thus, the basic idea of our approach is, only when our learning fails on a cut, we create a new cut by splitting the exactly failed partitions into small partitions according to the loop structure.

In the following, we present our algorithm for identifying a good cut. The details are as shown in Algorithm 3. The algorithm takes the program state SP and the program $\mathcal{P}$ as inputs. It maintains three variables: C is the current cut, *expanded* is the set of transitions in S which have been expanded, and *visited* are cuts which have been tried before. C is initialized as the set of outgoing transitions from *init*. Note that in our setting, there is only one transition from *init* to the start of the loop. The loop from line 3 to 16 tries different cuts each time to learn loop-invariants. At line 4, we first check whether the cut C has been tried before. If it is, we report failure. Otherwise, we add C into *visited*. At line 7, we attempt to generate a loop invariant using classification. If it is successful (i.e., satisfying the condition at line 8), we return the candidate. Otherwise, for each transition (s, s') in fts which corresponds to a partition which cannot be classified, we replace (s, s') with all outgoing transitions from s'. Intuitively, this is so that the partition corresponding to (s, s') is further partitioned. There are two

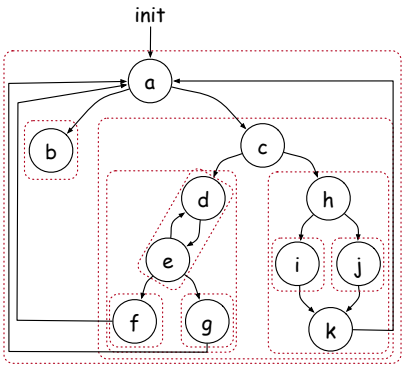


Figure 2: An abstract example

exceptions though. One is that s' has no outgoing transition, in which case we report failure. The other is that (s, s') is in *expanded* already, in which case, we do not add outgoing transitions of s'. Intuitively, this prevents trying the same cut again.

In fact, the classification algorithms can also effect the cut returned by Algorithm 3. On one hand, if there is a powerful classification algorithm that can classify any separable data, Algorithm 2 would generate a perfect classifer and thus no program structure needs to be uncovered. So Algorithm 3 always return the initial cut as the result. On the other hand, if a poor classification method is applied, the program structure needs to be unwinded very deeply before we can prefect classify the associated samples. To show the strength of our approach, we only adopt *SVM* and *SVM-I* as the learning algorithms.

Example 4.2. For the example in Figure 1a, the algorithm works as follows if *SVM-I* classification algorithm is adopted. Initially C is $\{(1,2)\}$, which means there is only one partition SP. As shown above, program states in SP are inseparable using the *SVM-I* algorithm, and thus we replace $(1,2)$ with two outgoing transitions from 2, i.e., $(2,10)$ and $(2,3)$. Thus, C becomes $\{(2,10),(2,3)\}$. Applying Algorithm 2, we succeed in learning a classifier $x < 2*N+4$ for the partition associated with transition $(2,10)$ and a classifier $x \geq y \wedge x+y \leq 2*N+2$ for the one associated with $(2,3)$. So the generated loop-invariant is: $(y < 0 \wedge x < 2*N+4) \vee (y \geq 0 \wedge x \geq y \wedge x+y \leq 2*N+2)$.

By contrast, if *SVM* classification is adopted, the partition associated with the transition $(2,3)$ can not be learned. Thus two outgoing transitions $(3,4)$ and $(3,6)$ replace $(2,3)$ in the cut. Now, C becomes $\{(2,10),(3,4),(3,6)\}$, with which our approach can generate exactly the same invariant as in Section 3.

In addition, if we instead apply the approach [48], we are unable to learn any candidate. If we apply the approach in [36] which partitions program states for every execution path in the loop, we may learn a loop invariant with three clauses as is shown in Section 3, if sufficient samples are provided.

We use the abstract example in Figure 2 to show that our approach can handle nested loops. Assume the current cut C contains the transition (d,e) and we fail to generate a classifier for the partition corresponding to transition (d,e). We replace (d,e) in C with transition (e,d), (e,f), and (e,g). Furthermore, if we fail again to generate a classifier for the partition corresponding to transition (e,d), since (e,d) has been added into *expanded*, we end up with a cut containing (e,f) and (e,g) but not (e,d).

A close look at the algorithm shows that it only leverages the structure of the nested loop in its first iteration. For instance, if an inner loop head has three outgoing transitions t_0, t_1 and t_2, assume there exist a state which visits t_1, t_2 and t_0 successively and a state which visits t_1 and t_0 consecutively. They are always parititioned in the same categories, as both of them encounter the transition t_1 first. However, a better solution would distinguish these two states. To the best of our knowledge, there are two ways to handle this. One is to distinguish all the different transition in the inner loop, which is equally to enumerate all the paths in a loop and thus is impractical for most loops with unbounded number of iterations. The other is to apply the same approach to the inner loop to obtain the inner loop-invariant, with which we can handle the (outer) loop later. But the problem with this strategy is we need to know the precondition and postcondition for the inner loop.

In the following, we briefly discuss the complexity of Algorithm 3. It can be easily proven that Algorithm 3 always terminates if Algorithm 1 always terminates. Intuitively, *expanded* never decreases during each iteration and its size is bounded by the number of transition in the program (which is finite). If *expanded* is not increased during one iteration, every transition in fts must be in *expanded*, in which case the current C remains the same and thus the algorithm terminates at line 5.

5 EVALUATION

We have implemented our approach as an extension of the open source Zilu toolkit developed initially in [36]. In the following, we refer to our version as Zimu [1]. It is written with a combination of C++ and shell codes (for invoking external tools). Our implementation uses LibSVM [14] as a primitive classification engine. For verifying candidate loop-invariants, we modify the KLEE project [13] to symbolically execute C programs prior to invoking Z3 [20] for checking the satisfiability of condition (4), (5) and (6). Note that integer-type variables in programs are encoded as integers in Z3 (i.e., not as bit vectors).

In order to check the effectiveness of our approach in learning disjunctive loop invariants, we choose benchmark C programs from multiple sources, such as previous publications (e.g., [22–25, 27, 33, 36]) and the software verification competitions 2017 [8]. Due to the few number of benchmark programs which require disjunctive programs reported in previous publications, we additionally construct 5 simple programs, including '`testb`', '`testb_ex`', '`testc`', '`testc_ex`' and '`zimu`' (the program in Figure 1a). We have a total of 76 programs. All of them are available at [1]. Unless otherwise stated, all the experiments are conducted using x64 Ubuntu 16.04.2 (kernel 4.8.0-58-generic) with 3.60 GHz Intel Core i7 and 32G DDR3. For execution, we limit the maximal number of guess-and-check iterations to 2 in Zimu. Due to randomness in generating the initial set of samples, each experiment is executed for 5 times and we report the medium as the result. In the following, we investigate the following research questions (RQ).

Table 2: Experiment results on "simple" programs

invariant type	benchmark	ZILU$_1$	ZIMU
linear	02, 03, 07, 08, 11, 14, 25, 26-29, 32, 33, 35, 47-50	✓	✓
conjunctive	04, 06, 09, 10, 13, 15-20, 22-24, 30, 34, 36-39, 41-43	✓	✓

Table 3: Experiment results on programs with branches or disjunctive postcondition

	ZILU$_1$		ZIMU			ZILU$_f$			ICE-CS		ICE-DT-e		ICE-DT-p		InvGen		CPAChecker	
program	✓✗	time	✓✗	time	$\sharp(c)$	✓✗	time	$\sharp(c)$	✓✗	time	✓✗	time	✓✗	time	✓✗	time	✓✗	time
benchmarks with non-disjunctive invariants																		
add	✓	10.06	✓	11.71	1	✓	9.20	2	✗	10.73	✗	TO	✗	TO	✗	0.08	✓	1.44
apractic	✓	4.22	✓	6.29	1	✓	5.69	3	✓	1.09	✓	1.36	✓	1.01	✗	0.05	✓	1.55
cav	✓	5.40	✓	9.86	2	✓	5.62	4	✗	TO	✓	23.83	✗	TO	✗	0.92	✓	153.20
dillig02	✓	8.41	✓	7.80	1	✓	7.33	3	✓	0.56	✓	0.75	✓	0.69	✓	0.04	✓	1.40
dillig05	✓	8.10	✓	10.81	1	✗	TO	-	✓	0.49	✓	0.71	✓	0.67	✓	0.06	✓	1.37
dillig08	✓	5.43	✓	5.56	1	✗	17.78	-	✓	0.50	✓	0.68	✓	0.65	✓	0.08	✓	1.59
trex2	✓	5.22	✓	4.96	1	✓	4.86	4	✓	0.69	✓	0.86	✓	0.87	✗	0.08	✓	1.48
trex3	✓	24.36	✓	14.44	1	✓	5.12	6	✓	3.46	✓	0.98	✓	0.90	✗	0.20	✓	1.46
zilu05	✓	74.65	✓	27.73	2	✓	6.52	3	✓	0.89	✓	0.93	✓	0.86	✗	0.06	✓	1.53
zilu12	✓	6.86	✓	6.58	1	✓	6.92	3	✓	0.54	✓	0.69	✓	0.66	✓	0.08	✓	1.30
zilu44	✓	4.59	✓	4.64	1	✓	5.16	4	✓	0.74	✓	1.26	✓	1.01	✗	0.07	✓	1.41
benchmarks with disjunctive invariants																		
dillig13	✗	TO	✗	TO	-	✗	TO	-	✗	TO	✗	TO	✗	TO	✗	0.06	✓	1.39
dillig19	✗	TO	✓	62.87	3	✓	40.05	3	✓	44.23	✓	2.95	✓	1.80	✗	0.07	✓	1.58
dillig35	✗	TO	✓	23.74	3	✓	6.96	3	✓	0.71	✓	0.77	✓	0.86	✗	0.06	✓	1.46
ex1	✗	TO	✓	16.10	3	✓	5.74	3	✗	TO	✗	TO	✗	TO	✗	0.39	✓	84.81
ex2_1	✗	TO	✓	17.32	3	✓	5.92	3	✗	TO	✗	TO	✗	TO	✗	0.41	✓	89.58
ex2_2	✗	TO	✓	26.05	3	✓	5.93	3	✗	TO	✗	TO	✗	TO	✗	0.19	✓	100.52
fig1	✗	TO	✓	75.61	2	✓	4.60	2	✓	0.69	✓	1.09	✓	10.9	✗	0.03	✓	1.28
fig2	✗	27.52	✓	13.98	2	✓	9.52	4	✗	12.51	✗	TO	✗	TO	✓	0.11	✓	1.39
fig3	✗	TO	✓	11.82	2	✓	4.59	3	✓	0.85	✓	0.95	✓	0.88	✗	0.05	✓	1.38
interp1	✗	TO	✓	7.96	2^{-1}	✓	28.59	2	✗	18.26	✗	TO	✗	TO	✓	0.09	✓	1.71
pldi082	✗	148.81	✓	11.98	2	✓	8.36	3	✗	67.67	✗	TO	✗	TO	✗	0.31	✓	1.72
pldi08	✗	TO	✓	56.78	2	✓	4.64	2	✓	0.62	✓	0.82	✓	0.78	✗	0.03	✓	1.57
popl	✗	TO	✓	20.84	3	✓	7.06	3	✗	128.31	✗	TO	✗	TO	✗	0.06	✓	30.81
popl_ex	✗	TO	✓	22.13	3	✓	9.12	4	✗	124.03	✗	TO	✗	TO	✗	0.21	✓	103.73
square	✗	TO	✓	9.86	1	✓	4.47	2	✗	14.31	✗	TO	✗	TO	✗	0.05	✗	1.25
sum1	✗	5.64	✓	17.42	3	✓	6.23	3	✓	2.49	✓	1.32	✓	1.21	✗	0.06	✓	1.69
sum4	✗	20.09	✓	7.11	3	✓	4.98	3	✓	2.92	✓	4.74	✓	2.04	✓	0.07	✓	2.22
sum4c	✗	TO	✓	18.38	3	✓	7.99	3	✓	2.64	✓	1.21	✓	1.10	✗	0.34	✓	1.31
sum4c_ex	✗	TO	✗	TO	-	✗	TO	-	✗	45.72	✗	TO	✗	TO	✗	0.04	✗	1.47
testb	✗	TO	✓	45.30	4	✓	4.85	4	✓	0.61	✓	0.81	✓	0.77	✗	0.03	✓	1.39
testb_ex	✗	TO	✓	37.80	4	✓	4.85	5	✓	0.57	✓	0.75	✓	0.73	✗	0.03	✓	1.49
testc	✗	TO	✓	113.77	6	✓	5.09	6	✓	1.14	✓	0.96	✓	0.91	✗	0.05	✓	1.45
testc_ex	✗	TO	✓	100.83	6	✓	4.95	6	✓	1.03	✓	0.90	✓	0.86	✗	0.06	✓	1.49
zimu	✗	TO	✓	11.98	2	✓	8.36	3	✗	50.57	✗	TO	✗	TO	✗	0.07	✓	1.72

RQ1: *Does using loop structures improve loop-invariant generation?* To answer this question, we compare ZIMU systematically with ZILU [36]. There are two settings of ZILU reported in [36]. One is denoted as ZILU$_1$ which can be regarded as a version of ZIMU without program state partitioning. The other is ZILU$_f$, which can be regarded as a version of ZIMU, in which there is one partition for every program path. Note that ZIMU$_f$ can not handle programs containing nested loops. The parameters in our experiments are set as follows. Selective sampling [36] is enabled in all three versions. The ratio between random sampling and selective sampling is set to be 5:1. Other parameters are set as default.

The experiment results are summarized in Table 2 and Table 3, where ✓ means that the Hoare triple is verified, ✗ means there is no conclusive results or false positives, *TO* means timeout after 6 minutes, and '-' means not available. In particular, Table 3 contains detailed results on programs whose loop body contains conditional branches or whose postcondition contains disjunctions, whereas Table 2 contains brief results on the rest of the programs. The reason for partitioning the programs is that ZIMU and ZILU$_1$ are expected

to have different results only for those programs in Table 3. Our experiment results confirm that ZIMU and ZILU$_1$ generate the same loop invariants and successfully verify those programs in Table 2. In Table 3, the first column shows benchmark programs, and the next 8 columns show details on verifying the Hoare triple through ZILU$_1$, ZIMU and ZILU$_f$. We show whether the verification is successful (37) and the number of disjunctive clauses ($\sharp(c)$) in the generated invariant. Note that for program `interp1`, one of the clauses is equivalent to false and it is shown as 2^{-1}.

The results show that ZIMU successfully verifies all 35 but 2 programs in Table 3, whereas ZILU$_1$ fails to verify 25 programs and ZILU$_f$ fails to verify 4 programs. In particular, ZILU$_1$ fails to verify any program which requires a disjunctive loop invariant. ZIMU fails to verify the two programs due to two reasons, '`sum4c_ex`' due to a polynomial invariant is needed for the program; while '`dillig13`' can not be handled due to a nondeterminism choice in the loop condition. We remark that the loops in these benchmark programs often contain nondeterministic choices, which are often used to model I/O environment (e.g., an external function call). In general, our approach works as long as the non-determinism is irrelevant to whether the post-condition is to be satisfied or not and the program state partitions are disjoint. However, if the loop contains a nondeterministic branch and a program state may nondeterministically either of the branches, the same state may be contained in both partitions if both branching transitions are in the cut. This undermines the proof of Theorem . In such a case, the disjunction of the classifiers learned for each partition (conjuncted with the path condition) is a weaker constraint, which may still work as a candidate loop invariants. This is why although there are several benchmark programs contain nondeterminism but ZIMU only fails to verify one of them.

A close investigation shows that ZILU$_f$ fails to verify the two programs ('`dillig05`' and '`dillig08`') which ZILU$_1$ and ZIMU successfully verify. This is also due to nondeterminism. For these two programs, non-deterministic choices are present in the loop body. As only linear/conjunctive invariants are needed for both cases, ZILU can handle these invariants while ZIMU also successfully learn the invariants before unwinding the inner structure of the loops. The difference between ZIMU and ZILU$_f$ is also shown in the number of clauses in the generated invariant. For 17 (out of 31) programs, ZIMU leans an invariant which has less clauses than that learned by ZILU$_f$. Given that having a simpler loop invariant is often helpful [2], we consider ZIMU to be better than ZILU$_f$ (i.e., verifies more programs and learns simpler invariants).

RQ2: *Does ZIMU outperform existing state-of-the-art tools on verifying these programs?* Ideally, we would like to compare with those tools reported in [23, 24, 44–48]. Unfortunately, many of them are not maintained. We instead compare ZIMU with three state-of-the-art tools on loop invariant generation and program verification in general. ICE [23] is a framework which learns loop invariants through a guess-and-check approach using examples, counter-examples, and implications. The original ICE implementation (ICE-CS) applies constraint solvers to find the candidate invariants. Later, the authors adopted decision tree learning algorithms to classify samples with different labels, and developed two versions of ICE with decision tree algorithms. One is ICE-DT-entropy (denoted as ICE-DT-e in the table) which chooses attributes based on entropy loss. The other is ICE-DT-penalty (denoted as ICE-DT-p in the table), which chooses attributes by penalizing cutting implications. We show the results of three different learning algorithms in ICE in Table 3 and refer the reader to [24] for details. We remark that the experiments with ICE are conducted with a different machine as its implementation only supports Windows system. Different environment may effect the learning time but not the learning results. The platform is x64 Windows 7 Ultimate SP1 with 2.40GHz Intel Core i7 and 16G DDR3, which is less powerful than the machine used for other experiments. We thus set the timeout to 10 minutes as compensation. InvGen is a tool which generates linear arithmetic invariants, using a combination of static and dynamic analysis techniques [29]. Lastly, CPAChecker [11] is a state-of-the-art program verifier. CPAChecker is on constant upgrading. The version we use is the one used for SV-COMP 2017 [8]. Note that CPAChecker supports a variety of verification methods and it is configured in the exact same way as in SV-COMP 2017[1]. For each tool, we show the verification results and the time taken for the verification.

The results are shown in the last 5 columns of Table 3. We observe that different learning algorithms implemented in ICE have similar performance. ICE-CS fails to verify 14 (out of 35) programs, ICE-DT-e fails to verify 13 programs and ICE-DE-p fails to verify 14 programs. In comparison, InvGen fails to verify 28 programs and CPAChecker impressively manages to verify all but 2 programs. CPAChecker's performance is not surprising given it uses techniques which go beyond loop invariant generation. In particular, one example is adjustable predicate analysis [37], which combines counterexample-guided abstraction refinement (CEGAR), lazy predicate abstraction, interpolation-based refinement, and large-block encoding. We are happy to take note that ZIMU verifies one example which even CPAChecker fails to verify (i.e., `square`).

RQ3: *Does ZIMU incur significant overhead?* To answer this question, we measure the time taken for each program by different tools and settings. We remark that the result on comparing the verification time should be taken with a grain of salt as these tools are based on different verification methods. Though ICE was experimented using a less powerful machine, it is proven to be efficient timewise nonetheless. Overall, ZIMU uses more time (from seconds to two minutes) than ZILU, ICE, InvGen and CPAChecker. This is because ZIMU takes many iterations to sample, learn and verify candidate invariants. Furthermore, it incrementally explores the program structure trying to learn a good cut. However, given that ZIMU, in all cases, manages to finish with two minutes, we consider that it is reasonably efficiently. We do notice that CPAChecker may take considerably longer time for verifying some programs. This is probably due to a different verification engine is applied for those programs after the initial one fails.

[1]We are thankful for the help from a researcher in the SV-COMP evaluation team.

6 RELATED WORK

Many approaches have been proposed for the loop-invariant generation. Examples include those based on abstraction interpretation [19, 35, 40], those based on counterexample-guided abstraction refinement [5, 16, 31] or interpolation [30, 38, 39], and those based on constraint solving and logical inference [17, 22, 28, 29]. In this following, we review those most related to our work.

Machine learning based approaches This work is related to loop-invariant generation using machine learning techniques. Besides those mentioned previous sections, in [44], the authors proposed a framework for generating invariants based on randomized search. In particular, their approach has two phases. In the search phase, randomized is used to discover candidate invariants. A checker is then used to either prove or refute the candidate in the validate phase. In [48], the authors proposed to generate samples through constraint solving and learn loop-invariants based on *SVM* classification. In [47], the authors proposed to apply PAC learning techniques for invariant generation. It has been demonstrated that their approach may learn invariants in the form of arbitrary boolean combinations of a given set of propositions (under certain assumptions). In [46], the authors developed a guess-and-check algorithm to generate invariants in the form of algebraic equations. It learns invariants in the form of polynomial equations by operating the null space operation on matrix. In [23], Pranav Garg *et. al* proposed to synthesize invariants by learning from implications along with positive and negative samples. They further extend their approach by modifying existing decision tree classification algorithm with heuristics adopted from [24]. These approaches apply advanced learning algorithms for generating a complicated classifier based on program states, while our approach solves the problem by leveraging the program structure information.

Inferring Disjunctive Invariants This work is related to existing approaches on inferring disjunctive invariants for loop-containing programs, some of which may also discover the invariants for proving some of our benchmarks. Template-based techniques [10, 17, 29] can find precise invariants if the user provides appropriate disjunctive templates. They are not fully automatic as the shape of the desired invariant needs to be specified in advance. Furthermore, their applicability is limited by the lack of efficient algorithms for solving complex constraints, if users provide templates containing complex constraints. The technique described in [25] uses counterexample guided abstraction refinement to tune widening strategies in an abstract interpretation framework. [26] models invariant generation task as probabilistic inference, and apply learning techniques to infer disjunctive invariants. Their technique is not guaranteed to converge, and it is difficult to characterize the class of loops for which it succeeds. Their contribution also includes an algorithm which extends the classical decision tree learning algorithm [42] to construct small decision trees using statistical measures.

Structure-based Invariant Learning This work is related to work on exploring program structures for code analysis tasks. The work in [45] presents a technique to decompose multi-phase loops into a semantically equivalent sequence of single-phase loops, each of which requires simple, conjunctive invariants. The idea of learning a complex invariant part by part is similar to our technique. There are two problems with their approach. One is that it does apply to any loop, i.e., a loop has to satisfy certain condition so that it can be split. The other is that their approach may split the loops unnecessarily. On the contrarily, our learning approach solves this problem by adaptively uncover the program structure.

7 CONCLUSION AND FUTURE WORK

In this work, we propose to generate disjunctive loop-invariant through learning based on program structural information. We present our cut-based invariant synthesis algorithm for learning invariants. We also highlight a heuristic cut searching technique according to program structure. For the future work, we would like to explore possibilities to generate loop-invariants through mining the program semantical information, which would complement the current work.

REFERENCES

[1] 2017. Zimu: An complex invariant inference framework based on learning with program structures. https://lunarplan.github.io. (2017).

[2] Aws Albarghouthi and Kenneth L. McMillan. 2013. Beautiful Interpolants. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 313–329. DOI: https://doi.org/10.1007/978-3-642-39799-8_22

[3] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and computation* 75, 2 (1987), 87–106.

[4] Domagoj Babic, Byron Cook, Alan J. Hu, and Zvonimir Rakamaric. 2013. Proving termination of nonlinear command sequences. *Formal Asp. Comput.* 25, 3 (2013), 389–403. DOI: https://doi.org/10.1007/s00165-012-0252-5

[5] Thomas Ball and Sriram K. Rajamani. 2001. The SLAM Toolkit. In *CAV*. 260–264.

[6] Thomas Ball and Sriram K Rajamani. 2002. The S LAM project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 1–3.

[7] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. 2009. Satisfiability Modulo Theories. *Handbook of satisfiability* 185 (2009), 825–885.

[8] Dirk Beyer. 2017. Software Verification with Validation of Results. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 331–349.

[9] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2007. The software model checker Blast. *STTT* 9, 5-6 (2007), 505–525. DOI: https://doi.org/10.1007/s10009-007-0044-z

[10] Dirk Beyer, Thomas A Henzinger, Rupak Majumdar, and Andrey Rybalchenko. 2007. Invariant synthesis for combined theories. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 378–394.

[11] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *CAV*. 184–190. DOI: https://doi.org/10.1007/978-3-642-22110-1_16

[12] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. 1992. A training algorithm for optimal margin classifiers. In *workshop on Computational learning theory*. ACM, 144–152.

[13] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[14] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)* 2, 3 (2011), 27.

[15] Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. 2015. Synthesising Interprocedural Bit-Precise Termination Proofs (T). In *ASE*. 53–64. DOI: https://doi.org/10.1109/ASE.2015.10

[16] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *Computer aided verification*. Springer, 154–169.

[17] Michael A Colón, Sriram Sankaranarayanan, and Henny B Sipma. 2003. Linear invariant generation using non-linear constraint solving. In *International Conference on Computer Aided Verification*. Springer, 420–432.

[18] Patrick Cousot and Radhia Cousot. 1977. 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*. ACM, 238–252.

[19] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *POPL*. ACM, 84–96.

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

[21] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. DOI: https://doi.org/10.1145/360933.360975

[22] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. 2013. Inductive invariant generation via abductive inference. In *OOPSLA*. 443–456.

[23] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 69–87. DOI: https://doi.org/10.1007/978-3-319-08867-9_5

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

[25] Bhargav S Gulavani, Supratik Chakraborty, Aditya V Nori, and Sriram K Rajamani. 2008. Automatically refining abstract interpretations. In *TACAS*. Springer, 443–458.

[26] Sumit Gulwani and Nebojsa Jojic. 2007. Program verification as probabilistic inference. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 277–289.

[27] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2008. Program analysis as constraint solving. In *PLDI*. 281–292. DOI: https://doi.org/10.1145/1375581.1375616

[28] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2009. Constraint-Based Invariant Inference over Predicate Abstraction. In *Proceedings of 10th International Conference on Verification, Model Checking, and Abstract Interpretation*. 120–135.

[29] Ashutosh Gupta and Andrey Rybalchenko. 2009. InvGen: An Efficient Invariant Generator. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. 634–640. DOI: https://doi.org/10.1007/978-3-642-02658-4_48

[30] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. 2004. Abstractions from proofs. In *POPL*. 232–244.

[31] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2003. Software verification with BLAST. In *Model Checking Software*. Springer, 235–239.

[32] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[33] Bertrand Jeannet. 2010. Interproc analyzer for recursive programs with numerical variables. *http://pop-art.inrialpes.fr/interproc/interprocweb.cgi* (2010), 06–11.

[34] Siddharth Krishna, Christian Puhrsch, and Thomas Wies. 2015. Learning Invariants using Decision Trees. *arXiv preprint arXiv:1501.04725* (2015).

[35] Vincent Laviron and Francesco Logozzo. 2009. SubPolyhedra: A (More) Scalable Approach to Infer Linear Inequalities. In *VMCAI*. 229–244.

[36] Jiaying Li, Li Li, Quang Loc Le, and Jun Sun. 2017. Automatic Loop-invariant Generation and Refinement through Selective Sampling. In *ASE*. to appear.

[37] Stefan Löwe and Philipp Wendler. 2012. CPAchecker with Adjustable Predicate Analysis - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 528–530. DOI: https://doi.org/10.1007/978-3-642-28756-5_40

[38] Kenneth L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *Computer Aided Verification*. 1–13.

[39] Kenneth L. McMillan. 2006. Lazy Abstraction with Interpolants. In *Computer Aided Verification*. 123–136.

[40] Antoine Miné. 2006. The octagon abstract domain. *Higher-Order and Symbolic Computation* 19, 1 (2006), 31–100.

[41] John Platt and others. 1998. Sequential minimal optimization: A fast algorithm for training support vector machines. (1998).

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

[43] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 24, 3 (2002), 217–298.

[44] Rahul Sharma and Alex Aiken. 2014. From invariant checking to invariant inference using randomized search. In *Computer Aided Verification*. Springer, 88–105.

[45] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Simplifying Loop Invariant Generation Using Splitter Predicates. In *CAV*. 703–719. DOI: https://doi.org/10.1007/978-3-642-22110-1_57

[46] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013. A Data Driven Approach for Algebraic Loop Invariants. In *ESOP*. 574–592. DOI: https://doi.org/10.1007/978-3-642-37036-6_31

[47] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. 2013. Verification as Learning Geometric Concepts. In *Static Analysis Symposium*. 388–411.

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