Loop Invariant Generation through Active Learning

Jiaying Li, Li Li, Le Guang Loc, Jun Sun

Singapore University of Technology and Design
jiaying_li@mymail.sutd.edu.sg
{li_li,guangloc_le,sunjun}@sutd.edu.sg

Abstract. Loop invariant generation is one of the fundamental problems in program analysis and verification. In this work, we propose an automatic generation method for inductive loop invariants through iterations of runtime sampling, machine learning and constraint solving. In each iteration, our method first collects real data at runtime based on selective sampling. Then, based on their satisfaction of the assumptions and assertions in the program, our method uses support vector machine to learn a loop invariant candidate, i.e., a conjunction of linear and polynomial constraints. Finally, if the candidate can be verified as the inductive loop invariant of the program, our method returns it directly and ends the generation process. Otherwise, the counter-example is used to refine the data sampling in the next iteration. The experiment evaluation shows that our method can be used to learn loop invariants effectively and automatically that cannot be learned by other loop invariant generation tools.

1 Introduction

In this work, we propose a method and a tool called ZILU for automatically learning loop invariants. ZILU works through through an iterative process combining sampling, active learning and verification.

2 Problem Definition and Solution Overview

In the following, we assume that a program contains a finite set of integer variable $\{x, y, z, \cdots\}$ and thus a program state is a valuation of the variables. A predicate on the variables is viewed as the maximum set of program states which satisfies the predicate. We use predicates and sets of program states interchangeably. Without loss of generality, we assume the input to ZILU is a Hoare triple

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

where Pre is the pre-condition, which should be satisfied before entering the loop; $Cond$ is the loop guard condition, which is the only way to enter or exit the loop $Body$; $Body$ is the loop body, in which we assume there is no *break* or *goto* statement which can jump out of the loop without checking $cond$; and $Post$ is the post-condition, which should be satisfied after the loop.

```
void ex1 (int x) {
    int y = 355;
    if (x > 46) x = 46;
    while (x <= 100) {
        if (x >= 46) {
            y = y+1;
        }
        x = x + 1;
    }
    assert(y==409);
}
```

Fig. 1. An example adopted from [**?**]

```
void ex2 () {
    lock=0;new=old+1;
    while (new!=old) {
        lock=1;old=new;
        if (foo(new)) {
            lock=0;new++;
        }
    }
    if (lock==0)
        error();
}
```

Fig. 2. An example adopted from [**?**]

For simplicity, we assume that $Body$ is a function such that $Body(s) = s'$ means that starting at a program state s, executing $Body$ would result in a program state s'. Furthermore, we write $Body(Pr)$ where Pr to denote the set $\{s' | \exists s \in Pr : Body(s) = s'\}$. The goal is thus to automatically obtain a loop invariant such that that the following conditions are satisfied.

$$Pre \implies Inv \qquad (1) Inv \implies Body(Inv \wedge Cond) \qquad (2) Inv \wedge \neg Cond \implies Post \qquad (3$$

Example 1. We use the two examples shown in Figure 1 and 2 to illustrate how our approach works. In $ex1$, the precondition of the loop is $y = 355 \wedge x \leq 46$ and the post-condition is $y = 409$. In $ex2$, the precondition is that $lock = 0 \wedge new = old + 1$ and the post-condition (necessary so that there is no error) is $lock = 1$. We remark that $foo(new)$ is an external function which *deterministically* returns either true or false, i.e., it returns true if new is even; otherwise, it returns false. We will discuss in Section how our approach would work if $foo(new)$ is non-deterministic.

Problem Definition In this work, we assume that given the Hoare triple, there is either a counterexample (i.e., a program state s such that $s \in Pre$ and executing the program from s results in failing $post$) or there exists an invariant satisfying (1) and (2) and (3). Furthermore, the invariant inv is a boolean formula over a linear inequality constraint of the form $ax + bx + \cdots \geq d$ where a, b, d are bounded integer constants; and inv contains no more than k such statements. We remark that such invariant is in general not convex and thus existing approaches on learning convex invariants do not work [].

Overview of Our Approach Our approach to solve the problem is illustrated in Figure **??**. Firstly, we randomly generate a set of program states right before the loop and test the program. Based on the testing results, we obtain program states which must or must not satisfy any invariant satisfying (1), (2) and (3). Secondly, we develop an algorithm for generating candidate invariants based classification techniques from the machine learning community. Thirdly, to overcome the limitation of the sampled program states, we adopt active learning techniques, in particular, selective sampling, to refine the candidate invariants. Lastly, we rely on constraint solving techniques to check whether the generate invariant satisfies (1) and (2) and (3). If it does, we report that

our approach is successful; otherwise, using the counterexamples generated by the constraint solvers, we repeat from the second step. In this following sections, we present details of each step.

3 Sampling

Before learning procedure, we need to gather some data from program. So in this step, we first sample a set S of program states in three ways, random sampling, selective sampling and counter-example sampling, according to the learning phases.

In this step, we sample, either randomly or using tools based on the idea of concolic testing [], a set T of program states and test the program starting with each program state s in T. We write $Body^*(s)$ to denote the set of program states which could be reached after executing zero or more iterations of the loop starting from s. We write $Body^*(T)$ to denote $\{s' | \exists s \in T \cdot s' \in Body^*(s)\}$. Furthermore, we write $s \Rightarrow s'$ to denote that starting with a program state s would result in state s' when the loop terminates.

We categorize program states in $Body^*(T)$ into four sets: C_T which stands for counter-example trace, P_T which stands for traces with positive labels, N_T which stands for traces with negative labels and U_T which stands for traces with unknown labels. They can be judged by the following rules:

- Set C_T is $\{s \in Body^*(T) | s \in Pre \wedge s \Rightarrow s' \wedge s' \notin Post\}$;
- Set P_T is $\{s \in Body^*(T) | s \in Pre \wedge s \Rightarrow s' \wedge s' \in Post\}$;
- Set N_T is $\{s \in Body^*(T) | s \notin Pre \wedge s \Rightarrow s' \wedge s' \notin Post\}$;
- Set U_T is $\{s \in Body^*(T) | s \notin Pre \wedge s \Rightarrow s' \wedge s' \in Post\}$;

We remark that anytime a program state in C_T is identified, a counter-example is found and ZILU reports that verification is failed immediately. Otherwise, because Inv must satisfy (1),(2) and (3), we know that $P_T \subseteq Inv$ and $N_T \cap Inv = \emptyset$. The program states in U_T may or not may be in Inv. If we know that a program state $s \in U_T$ is in Inv, $Body^*(s) \subseteq Inv$.

3.1 Random Sampling

Random Sampling is applied all along our learning process. It is a quite intuitive sampling approach, in which each sample is chosen randomly and entirely by chance, such that each sample has the same probability of being chosen at any stage during the sampling process. At the begging of our framework, we use sampling mechanism completely as we have no idea of sample information then. In the subsequent iterations, we also use this sampling method partially to avoid the bias resulted from previous sampling iteration.

3.2 Selective Sampling (Active Learning)

When we have a guess of loop invariants, we can apply selective sampling approach to find more useful samples. Actually we apply this sampling method all along the

Algorithm 1: Algorithm $activeLearning$

Input: F^+ and F^-
Output: a classifier for F^+ and F^-

```
1 let old be null;
2 while true do
3     let f = classify(F+, F−);
4     if f is identical to old then
5         return f;
6     let old = f;
7     let sam be a set of samples computed by selective sampling;
8     test the program and update F+ and F− accordingly;
```

learning procedure except the first iteration. Due to the limited set of samples we have (which is often referred to as labeled samples in the machine learning community), the guessed classifier obtained from previous iteration might be far from being correct. In fact, without labeled samples which are right on the boundary of the 'actual' classifier, it is very unlikely that we would find it. Intuitively, in order to get the 'actual' classifier, we would require samples which would distinguish the actual one from any nearby one. This problem has been discussed and addressed in the machine learning community using active learning and selective sampling [**?**].

The concept of active learning or selective sampling refers to the approaches that aim at reducing the labeling effort by selecting only the most informative samples to be labeled. SVM selective sampling techniques have been proven effective in achieving a high accuracy with fewer examples in many applications [**?**,**?**]. The basic idea of selective sampling is that at each round, we select the samples that are the closest to the classification boundary so that they are the most difficult to classify and the most informative to label. Since an SVM classification function is represented by support vectors which are the samples closest to the boundary, this selective sampling effectively learns an accurate function with fewer labeled data [**?**]. In our setting, this means that we should sample a program state right by the classifier and test the program with that state to label that feature vector so that the classifier would be improved.

Algorithm 1 presents details on how active learning is implemented in ZILU. At line 2, we obtain a classifier based on Algorithm **??**. We compare the newly obtained classifier with the previous one at line 4, if they are identical, we return the classifier; otherwise we apply selective sampling so that we can generate additional labeled samples for improving the classifier. In particular, at line 5, we apply standard techniques [**?**] to select the most informative sample. Notice that in our setting, the most informative samples are those which are exactly on the lines and therefore can be obtained by solving an equation system. At line 8, we test the program with the newly generated samples so as to label them accordingly.

In our implementation, after getting a classifier, which is usually a single polynomials or conjunction or disjunction of polynomials, we can get some solutions of these

polynomials using **GSL** (GNU Scientific Library). Then we use these solutions or the points near these solutions as sampling points.

3.3 Counter-Example Sampling

Counter-Example sampling is applied after failure of invariant candidate verification (which is described in details in section 5). After getting an invariant candidate, we try to check it using concolic testing [] and constraint solving. When we fails to validate, the constraint solver could provide us with counter-examples, which directly refute our invariant candidate. And as a result, it is quite useful for the invariant candidate refinement in next learning procedure.

4 Labeling

With the sample set S from the last step, we test the program starting with each program state s in S. We write $Body^*(s)$ to denote the set of program states which could be reached after executing zero or more iterations of the loop starting from s. So if there is a trace $Trace\{s_0, s_1, ..., s_n\}$, then $s_i \in Body^*(s_0)\ for i \in [0...n]$. We write $Body^*(S)$ to denote $\{s' | \exists s \in S \cdot s' \in Body^*(s)\}$. Furthermore, we write $s \Rightarrow\Rightarrow s'$ to denote that starting with a program state s would result in state s' when the loop terminates.

Positive State, Negative State & Implication State These three concepts are introduced in [**?**]. In this paper, we use predicates and sets of states interchangeably. Let C be a candidate invariant.

From equation (1) we know, for an invariant Inv, any state that satisfies Pre also satisfies Inv. We call any state that must be satisfied by an actual invariant a good state.

Now consider equation (2). A pair (s, t) satisfies the property that s satisfies $Cond$ and if the execution of S is started in state s then S can terminate in state t. Since an actual invariant Inv is inductive, it should satisfy $s \in Inv \Rightarrow t \in Inv$. Hence, a pair (s, t) satisfying $s \in C \wedge t \notin \mathcal{C}$ proves C is not an invariant.

Finally, consider equation (3). The 'existence of a state $s \in C \wedge \neg B \wedge \neg Post$ proves C is inadequate to discharge the postcondition. We call a state s which satisfies $\not{B} \wedge \not{Post}$ a bad state.

Positive Trace, Negative Trace & Implication Trace In our approach, we assume $\{s_0, s_1, s_2, ..., s_i, ..., s_n\}$ is a chain of states in the target program, where s_0 is the initial state before entering the loop, and s_i is a state just after the loop has iterated i times in the program. We assume s_n satisfies $\neg B$ so it is the state that can jump out the loop body.

For a state chain $\{s_0, s_1, s_2, ..., s_i, ..., s_n\}$, if s_0 satisfies Pre, and s_n satisfies $Post$, we say this is a good state chain, Because if state s_0 satisfy Pre, s_0 is a good state that must satisfy Inv, according to equation 1. Furthermore, according to equation 2, all of $\{s_1, s_2, ..., s_i, ..., s_n\}$ are good states. So now we can get a good state chain $\{s_0, s_1, s_2, ..., s_i, ..., s_n\}$.

On the contrary, for a state chain $\{s_0, s_1, s_2, ..., s_i, ..., s_n\}$, if s_0 satisfies $\neg Pre$, and s_n satisfies $\neg Post$, we say this is a bad state chain, which means all the states in this chain should be bad states.

We categorize program states in $Body^*(S)$ into four sets: C_S which stands for counter-example trace, P_S which stands for traces with positive labels, N_S which stands for traces with negative labels and U_S which stands for traces with unknown labels.

They can be judged according to Table 4:

Table 1. Trace Labeling Table

$Trace\{s_0, s_1, ..., s_n\}$	$s_n \in Post$	$s_n \in \neg Post$
$s_o \in Pre$	$Trace \in P_S$	$Trace \in C_S$
$s_0 \in \neg Pre$	$Trace \in U_S$	$Trace \in N_S$

Actually for an arbitrary state chain there are also two other possibilities we have not mentioned yet. One is a chain begins with a state s_0 that satisfies Pre but end with a state s_n that satisfies $\neg Post$, this is a counterexample to disprove the program. That means there is something wrong with at least one of precondition, loop condition, loop body or postcondition. We need to find out what happens and update the program, after which we can reapply our approach to learn loop invariants. The other case is a chain begin with a state s_0 that satisfies $\neg Pre$ but end with a state s_n that satisfies $Post$. Under this condition, we could not justify whether s_0 and s_n satisfy invariants or not, not to mention other states $\{s_1, s_2, ..., s_i, ..., s_{n-1}\}$. The only thing we can ensure is this is an implication chain. In total, we can have table. **??**.

Among all these possibilities, we can get more samples than previous approaches can with executing program just once. So with the sample information, the learner can learn an as good invariant as, if not better than, the previous approaches, as it can utilize more information to do invariant learning task. This also implies our approach can get convergence faster than before.

Example 2.

Proposition 1. *Algorithm $activeLearning$ always eventually terminates.* □

Example 3.

5 Classification

After sampling and labeling, we obtain some program states must be in Inv and some must not. Thus, any candidate invariant must be able to perfectly classify these states. We apply classification techniques from the machine learning community to obtain classifiers as candidate invariants. Due to different technique can results in different forms of classifiers, we apply SVM and some of its derivatives on our training data.

Algorithm 2: Algorithm $overall$

Input: Pre, $Cond$, $Body$, $Post$
Output: an invariant which completes the proof or a counterexample

```
let T be a set of random samples;
while true do
    test the program for each sample in T;
    if a state s in C is identified then
        return s as a counterexample;
    let P, N and NP be the respective sets accordingly;
    let Inv_u = activeLearning(P, N ∪ NP);
    let Inv_o = activeLearning(P ∪ NP, N);
    let Inv_s = activeLearning(P, N);
    for each Inv in {Inv_u, Inv_o, Inv_s} do
        if (1) or (2) or (3) is not satisfied then
            add the counterexample into T;
        else
            return Inv as the proof;
```

5.1 Linear SVM

5.2 Polynomial SVM

5.3 Conjunctive SVM

In the following, we present how we obtain a classifier automatically using SVM. SVM is a supervised machine learning algorithm for classification and regression analysis. We use its binary classification functionality. Mathematically, the binary classification functionality of (linear) SVM works as follows. Given two sets of feature vectors F^+ and F^-, it generates, if there is any, a linear constraint in the form of $ax + by + \cdots \geq d$ where x and y are feature values and a, b, d are constants, such that every state $s \in F^+$ satisfies the constraint and every state $s' \in F^-$ fails the constraint. In this work, we always choose the *optimal margin classifier* (see the definition in [**?**]) if possible. This half space could be seen as the strongest witness why the two data states are different. In the following, we write $svm(F^+, F^-)$ to denote the function which returns a linear classifier

If, however, F^+ and F^- cannot be perfectly classified by one half space only, a more complicated function f must be adopted. For instance, if there is a classifier in the form of conjunctive of multiple half spaces, the algorithm presented in [**?**] can be used to identify such a classifier.

6 Verification

Given a learned predicate Inv, we verify whether constraint (1), (2) and (3) are satisfied using symbolic execution and constraint solving. First we separate the given loop

program with the leaned predicate into three loop-free programs, which corresponds to constraints (1), (2) and (3). After compiling these programs, we apply KLEE on each of them to get all the path condition. KLEE If all of them are satisfied, we successfully verify the program. Otherwise, if any of them is violated, the counterexample obtained is added to the set of sample X, (named as counter-example sampling in section 3) which is then tested, categorized, used for active learning accordingly. The overall algorithm is presented in Figure 2.

We remark that we learn three classifiers as candidates for the loop invariant: U, OU, O such that

- U classifies states in P and those in $N \cup NP$.
- O classifies states in N and those in $P \cup NP$.
- OU classifies states in P and N;

Intuitively, U would be an under-approximation of Inv (by assuming states in NP does not satisfy Inv); O would be an over-approximation of Inv (by assuming states in NP does satisfy Inv); and OU would be an safe-approximation of Inv (by using states which we are certain whether they are in Inv or not).

Example 4.

Theorem 1. *Algorithm $overall$ always eventually terminates and it is correct.* □

7 Experiments

8 Related Work

9 Conclusion

Limitation and Potential Remedies We remark that in theory, we could learn non-linear classifier using methods like SVM with kernel methods []. Nonetheless, due to the limitation of proving capability and tools with regards to non-linear constraints, we leave those to our future work. Furthermore, we assume there is a bound k on the number of clauses in the variant. In practice, we would expect (refer to empirical evidence in Section 7) often k is of a small value.

References