Loop Invariant Generation through Active Learning

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

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

Abstract. In this work, we propose an automatic generation method for loop invariants through iterations of selective sampling, machine learning and constraint solving. In each iteration,

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

where *Pre* is the pre-condition; *cond* is the loop guard condition; *Body* is the loop body; and *Post* is the post-condition. 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 a in program state s'. Furthermore, we write $Body(Pr)$ where *Pr* to denote the set $\{s' \mid \exists s \in Pr : Body(s) = s'\}$. The goal is thus to automatically obtain a loop invariant such that that the following conditions are satisfied.

$$Pre \Rightarrow Inv \quad (1)$$
$$Inv' \Rightarrow Body(Inv \wedge Cond) \quad (2)$$
$$Inv \wedge \neg Cond \Rightarrow Post \quad (3)$$

where Inv' is the predicate obtained by replacing every variable in *Inv* with its primed version, denoting the set of program states after executing *Body*.

```
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 [1]

```
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 [2]

Example 1. We use the two examples shown in Figure 1 and 2 to illustrate how our approach works. In *ex*1, the precondition of the loop is $y = 355 \wedge x \leq 46$ and the post-condition is $y = 409$. In *ex*2, 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

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 $\{s' \mid \exists s \in T \cdot s' \in Body^*(s)\}$. 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:

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

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

Example 2.

4 Classification

After sampling, 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.

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 [4]) 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 [4] can be used to identify such a classifier.

4.1 Clustering

Based on the above assumption, the linear constraints ϕ_i divide the space into at most 2^k convex regions. One simple example is shown in Figure **??**. All program states in the same region must have the same label, i.e., whether it satisfies the invariant or not. Every pair of regions R_1 and R_2 must be linearly separable, i.e., there exists some linear constraint ϕ such that $R_1 \subseteq \phi$ and $R_2 \cap \phi = \varnothing$. Based on this observation, a naive

Algorithm 1: Algorithm *cluster*

Input: F^+ and F^-
Output: a set of clusters which are pairwise linearly separable

let $k = 1$;
while $k < K$ **do**
  apply k-means algorithm to cluster F^+ into k clusters $R_1^+, R_2^+, \cdots, R_k^+$;
  apply k-means algorithm to cluster F^- into k clusters $R_1^-, R_2^-, \cdots, R_k^-$;
  if *every pair of* R_i^+ *and* R_j^- *where* $1 \leq i \leq k$ *and* $1 \leq j \leq k$ *is linearly separable* **then**
    break;
  $k = k + 1$;
let *merged* be true;
while *merged* **do**
  let *merged* be false;
  for *each pair* R_i^+ *and* R_j^+ *where* $i \neq j$ **do**
    if $R_i^+ \cup R_j^+$ *is linearly separable from* R_i^- *for all* i **then**
      merge R_i^+ and R_j^+; set *merged* be true;
  for *each pair* R_i^- *and* R_j^- *where* $i \neq j$ **do**
    if $R_i^- \cup R_j^-$ *is linearly separable from* R_i^+ *for all* i **then**
      merge R_i^- and R_j^-; set *merged* be true;
return the clusters;

algorithm for clustering the states in F^+ and F^- would work as follows: given a k value such that $k \geq 2$, we randomly assign every state s in $F^+ \cup F^-$ to a region R_i as long as all states in the same region have the same label and every pair of regions remain linearly separable. We remark that if k is 2, there is only one clustering. In general, this algorithm is rather inefficient if you try all possible clustering. Intuitively, however, *'nearby'* states often belong to the same clustering and thus in the following algorithm, we first group near-by states and then identify the clustering.

Algorithm 1 works in two phases. In the first phase we apply k-means algorithm [] to group the data into linearly separable clusters. Notice that the states in F^+ and F^+ are clustered separably since all states in the same cluster must have the same label. In order to avoid too many clusters, in the second phase, we merge clusters into larger clusters.

Example 3.

Proposition 1. *cluster*(F^+, F^-) *returns two sets of clusters* $\{R_1^+, R_2^+, \cdots, R_m^+\}$ *and* $\{R_1^-, R_2^-, \cdots, R_n^-\}$ *such that every pair of* R_i^+ *and* R_j^- *where* $1 \leq i \leq k$ *and* $1 \leq j \leq k$ *is linearly separable.* □

4.2 Classification

After the last step, we obtain the clusters $R_1^+, R_2^+, \cdots, R_m^+$ and $R_1^-, R_2^-, \cdots, R_n^-$ (where $m \leq k$ and $n \leq k$). Next, we generate candidate classifiers. Our approach is based on the

Algorithm 2: Algorithm *classify*

Input: clusters $R_1^+, R_2^+, \cdots, R_m^+$ and $R_1^-, R_2^-, \cdots, R_n^-$
Output: a candidate invariant

```
1 let X be an empty sequence;
2 for each pair R_i^+ and R_j^- where 1 ≤ i ≤ m and 1 ≤ j ≤ n do
3     for each possible labeling of the clusters other than R_i^+ and R_j^- do
4         apply SVM to generate a linear classifier;
5         if there is a perfect classifier inv then
6             add inv into X;
7         if any two clusters with different labels can be separated by a classifier in X then
8             return the candidate invariant based on X;
```

observation that for each actual classifier (i.e., a clause in *inv*), there exists at least one way of re-labeling the clusters such that the classifier would classify all states correctly. For instance, Figure **??** shows the states can be re-labeled so that a linear constraint in Figure **??** becomes a classifier for all states. Based on the observation, we design Algorithm 2 to identify candidate invariant.

Firstly, we identify a set X of candidate linear constraints. For each pair of clusters with different labels, for any possible re-labeling of the other clusters, if there is a linear classifier which perfectly classifies all states after relabeling, we add the linear classifier into X. Whenever any two clusters can be separated by a classifier in X, we return a candidate invariant based on X which is in the following form.

$$\begin{array}{l}\bigwedge\{\phi \in X \mid R_1^+ \subseteq \phi\} \vee \\ \bigwedge\{\phi \in X \mid R_2^+ \subseteq \phi\} \vee \\ \cdots \\ \bigwedge\{\phi \in X \mid R_m^+ \subseteq \phi\}\end{array}$$

In the worst case, the loop from line 2 to 8 would iteration for $m * n * 2^{m+n-2}$ times.

Example 4.

Proposition 2. *Given two set of states F^+ and F^- such that there is a classifier in the assumed form, classify(F^+, F^-) always returns a perfect classifier.* □

5 Selective Sampling

Due to the limited set of samples we have (which is often referred to as labeled samples in the machine learning community), the classifier obtained above 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

Algorithm 3: 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;
```

from any nearby one. This problem has been discussed and addressed in the machine learning community using active learning and selective sampling [3].

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 [5, 6]. 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 [3]. 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 3 presents details on how active learning is implemented in ZILU. At line 2, we obtain a classifier based on Algorithm 2. 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 [3] 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.

Example 5.

Proposition 3. *Algorithm activeLearning always eventually terminates.* □

6 Verification

Given a learned predicate *Inv*, we verify whether constraint (1), (2) and (3) are satisfied using symbolic execution. 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

Algorithm 4: Algorithm *overall*

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

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

set of sample X, which is then tested, categorized, used for active learning accordingly. The overall algorithm is presented in Figure 4.

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 6.

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

7 Implementation and Evaluation

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

1. S. Gulwani and N. Jojic. Program verification as probabilistic inference. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 277–289, 2007.
2. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 58–70, 2002.
3. G. Schohn and D. Cohn. Less is more: Active learning with support vector machines. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*, pages 839–846, 2000.
4. R. Sharma, A. V. Nori, and A. Aiken. Interpolants as Classifiers. In *CAV*, pages 71–87, 2012.
5. S. Tong and E. Y. Chang. Support vector machine active learning for image retrieval. In *Proceedings of the 9th ACM International Conference on Multimedia 2001, Ottawa, Ontario, Canada, September 30 - October 5, 2001*, pages 107–118, 2001.
6. S. Tong and D. Koller. Support vector machine active learning with applications to text classification. *Journal of Machine Learning Research*, 2:45–66, 2001.