Loop Invariant Generation through Active Learning

Jiaying Li[1], Jun Sun[1], Li Li[1], Quang Loc Le[1] and Shang-Wei Lin[2]

[1] Singapore University of Technology and Design
[2] School of Computer Science and Engineering, Nanyang Technological University

Abstract. Loop invariant generation is important in program analysis and verification. In this work, we propose a technique for automatically loop invariant generation through a combination of active learning and verification. Given a Hoare triple of a program containing a loop, we start with randomly testing the program. We collect program states at run-time and categorize them based on whether they satisfy the invariant to be discovered. Next, classification techniques are employed to generate candidate loop invariants. In particular, we refine the candidates through active learning so as to overcome the lack of sampled program states. Only after the candidate invariant cannot be improved further through active learning, we verify whether a candidate can be used to prove the Hoare triple. If it cannot, the generated counterexample is used for re-classification and we repeat the above process. Furthermore, we show that by introducing path-sensitive learning, i.e., partitioning the program states according to program locations they visit and classifying each partition separately, we are able to learn disjunctive loop invariants. We have developed a prototype tool and applied it to verify a set of benchmark programs. The evaluation shows that our approach complements existing approaches.

1 Introduction

Automatic loop invariant generation is fundamental for program analysis. A loop invariant can be useful for software verification, compiler optimization, program understanding, etc. In the following, we first define the loop invariant generation problem and then briefly describe existing approaches and then our proposal. For simplicity, we assume that we are given a Hoare triple in the following form.

$$\begin{array}{ll} \{Pre\} & /\star \textit{Assumption} \star / \\ while(Cond)\{Body\} & /\star \textit{Loop Body} \star / \\ \{Post\} & /\star \textit{Assertion} \star / \end{array}$$

Assume that $V = \{x_1,x_2,\cdots,x_n\}$ is a finite set of program variables which are relevant to the loop body. Pre, $Cond$ and $Post$ are predicates constituted by variables in V.

Let $s = \{x_1 \mapsto v_1,\cdots,x_n \mapsto v_n\}$ be a valuation of V. 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 given the valuation. We thus write $s \in \phi$ to denote that ϕ is evaluated to $true$ given s. Otherwise, we write $s \notin \phi$. $Body$ is an imperative program which

updates the valuation of V. For simplicity, we assume that it is a deterministic function[3] on valuations of variables V, and write $Body(s)$ to denote the valuation of V after executing $Body$ given the variable valuation s. For convenience, $Body^i(s)$ where $i \geq 0$ is defined as follows: $Body^0(s) = s$ and $Body^{i+1}(s) = Body(Body^i(s))$.

The problem is to either prove the Hoare triple or disprove it. In order to prove the Hoare triple, we would like to find a loop invariant Inv which satisfies the following three conditions.

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

$$\forall s.\ s \in Inv \wedge Cond \implies Body(s) \in Inv \tag{2}$$

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

In order to disprove the Hoare triple, we would like to find a valuation s such that $s \models Pre$ and executing the loop until it terminates results in a valuation s' such that $s' \not\models Post$. For simplicity, we assume that the loop always terminates and refer the readers to [2,10] for extensive research on proving loop termination.

Many approaches have been proposed to solve this problem. For example, there are proposals based on abstraction interpretation [13,36,32], counterexample guided abstraction refinement [28,3,11], interpolation [27,34,35] and constraint solving and inference [25,12,24]. Recently, the authors of [49,48,47,45] proposed to automatically generate loop invariants based on random searching [45] as well as machine learning [49]. Their approaches start with randomly generating valuations of V (a.k.a. the samples) and categorize them into different groups, e.g., one containing those satisfying the loop invariant (if there is any) and another containing those not. Machine learning techniques are then used to generalize them in a certain form to obtain candidate loop invariants. The candidates are then checked using program verification techniques (like symbolic execution [38]) to see whether they satisfy the three conditions. If any of the conditions is violated, we obtain counterexamples in the form of variable valuations. For instance, given a candidate ϕ, if condition (1) is violated, a valuation $s \in (Pre \wedge \neg\phi)$ is generated, which proves that ϕ is not an invariant. With the new sample s, we can reclassify the samples to obtain a new candidate invariant. This guess-and-check process is repeated until either the Hoare triple is proved or disproved.

One problem with the guess-and-check approach is that its effectiveness is often limited by the samples which are generated randomly. In order to learn the right invariant through classification, often a large number of samples are necessary. Furthermore, often those samples right by the boundary between variable valuations which satisfy the actual invariant and those which do not must be sampled so that classification techniques would identify the right invariant. Obtaining those samples through random sampling is often hard. As a result, many iterations of guess-and-check are required. Another problem is that the kinds of loop invariants obtained through existing guess-and-check approaches [49,48,47,45] are often limited, e.g., conjunctive linear inequalities [49] or equalities [47]. Despite the approach presented in [23,46], learning disjunctive loop invariants remains a challenge.

[3] Our approach works as long as the non-determinism in $Body$ or $Cond$ is irrelevant to whether the postcondition is satisfied or not.

In this work, we propose a technique to improve the existing guess-and-check approaches [49,48,47,45]. Compared to the existing approaches, we make the following contributions. Firstly, we propose an active learning technique to overcome the limitation of random sampling. That is, the active learning technique allows us to automatically generate samples which are important in improving the quality of the candidate invariants so that we can improve the candidates prior to checking them during every guess-and-check iteration. As a result, we can reduce the number of guess-and-check iterations, or even completely in many cases. Secondly, our approach is designed to be extensible so that we can learn different kinds of invariants. For instance, we show that we can learn candidate invariants in the form of polynomial inequalities or their conjunctions using a different classification algorithm. Furthermore, we show that by introducing a *path-sensitive* learning through two steps: partitioning the samples according to the control locations they visit and classifying each partition separately, we are able to generate disjunctive invariants. Lastly, we implement our framework as a tool called ZILU (available at [1]) and compare it with state-of-the-art tools like Interproc [30] as well as CPAChecker [6].

The remainders of the paper are organized as follows. Section 2 presents an overview of our approach using an illustrative example. Section 3 shows how candidate loop invariants are generated and refined through active learning. Section 4 discusses our implementation and evaluates its effectiveness using a set of benchmark programs. Section 5 reviews related work and concludes.

2 The Overall Approach

Loop invariant generation using a guess-and-check approach is an iterative process of *data collection*, *guessing* (i.e., classification in this work) and *checking* (i.e., verification of the invariant candidate). In the following, we present how our approach works step-by-step and illustrate each step with simple examples.

Example 1. A few example Hoare triples are shown in Figure 1, where an `assume` statement captures the precondition and an `assert` statement captures the postcondition. The set of variables V for each program contains two integer-type ones: x and y. For simplicity, we write (a, b) where a and b are integer constants to denote the evaluation $\{x \mapsto a, y \mapsto b\}$. Further, we interpret integers in the programs as mathematical integers (i.e., they do not overflow). One example invariant which can be used to prove the Hoare triple is shown for each program. For instance, the Hoare triple shown in Figure 1(a) can be proven using a loop invariant: $x \leq y + 16$, whereas conjunctive or disjunctive invariants are necessary to prove the other Hoare triples. In the following, we show how we generate loop invariants for proving these Hoare triples.

The overall approach is shown as 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 the initial set of samples to learn the very first candidate for the loop invariant. In this work, we have two ways to generate random samples. One is that we generate random values for each variable in V based on its domain, assuming a uniform probabilistic distribution over all values in its domain. The other is that we

```
1  assume(x < y);
2  while(x < y){
3    if (x < 0) x := x + 7;
4    else x := x + 10;
5    if (y < 0) y := y − 10;
6    else y := y + 3;
7  }
8  assert(y ≤ x ≤ y + 16);
```

(a) Invariant: $x \leq y + 16$

```
1 assume(x > 0 ∨ y > 0);
2 while(x + y ≤ − 2){
3   if (x > 0){
4       x := x + 1;
5   } else {
6     y:=y + 1;
7   }
8 }
9 assert(x > 0 ∨ y > 0);
```

(b) Invariant: $x > 0 \vee y > 0$

```
1  assume(x = 1∧y = 0);
2  while(∗){
3    x:=x + y;
4    y:=y + 1;
5  }
6  assert(x ≥ y);
```

(c) Invariant: $y \geq 0 \wedge x \geq y$

```
1 assume(x < 0);
2 while(x < 0){
3     x = x + y;
4     y++;
5 }
6 assert(y > 0);
```

(d) Invariant: $x < 0 \vee y > 0$

Fig. 1: Example programs

apply an SMT solver [4,15] to generate valuations that satisfy Pre as well as those that fail Pre. These two ways are complementary. On one hand, without using a solver, we may not be able to generate valuations which satisfy Pre if Pre is very restrictive (or fail Pre if the negation of Pre is very restrictive). On the other hand, using a solver often generates biased valuations.

Next, for any valuation s in SP, we execute the program starting with initial variable valuation s and record the valuation of V after each iteration of the loop. We write $s \Rightarrow s'$ to denote that there exists $i \geq 0$ such that $s' = Body^i(s)$ and $Body^k(s) \in Cond$ for all $k \in [0, i)$. That is, 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 the four disjoint sets: CE, $Positive$, $Negative$ and NP. Intuitively, CE contains counterexamples which disprove the Hoare triple; $Positive$ contains those valuations of V which we know must satisfy any loop invariant which proves the Hoare triple; $Negative$ contains those valuations of V which we know must not satisfy any loop invariant which proves the Hoare triple; and NP contains the rest.

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

A valuation in $CE(SP)$ 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.

$$\begin{aligned} Positive(SP) = \{&s \in SP | \exists s_0, s_1 : SP. \\ &s_0 \in Pre \wedge s_0 \Rightarrow s \Rightarrow s_1 \wedge s_1 \notin Cond \wedge s_1 \in Post\} \end{aligned}$$

$Positive(SP)$ 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

Algorithm 1: Algorithm $zilu()$

```
1 let SP be a set of randomly generated valuations of V;
2 while not time out do
3     add all valuations s' such that s ⇒ s' for some s ∈ SP into SP;
4     call actL(SP) to generate a candidate invariant;
5     return "proved" if the program is verified with φ otherwise add the counterexample
      into SP;
```

s' which satisfies $Post$ when the loop terminates. Let Inv be any loop invariant which proves the Hoare triple. Because $s_0 \in Pre$, $s_0 \in Inv$ since Inv satisfies condition (1). Since Inv satisfies condition (2) and $Body(s_0) \in Inv$ if $Body(s_0) \in Cond$. By a simple induction, we prove $s \in Inv$.

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

$Negative(SP)$ is the set of valuations which violates Pre and becomes a valuation s' which violates $Post$ when the loop terminates. We show that $s \notin Inv$ for all Inv satisfying condition (1), (2) and (3). Assume that $s \in Inv$, by condition (2), s' must satisfy Inv through a simple induction. By condition (3), s' must satisfy $Post$, which contradicts the definition of $Negative(SP)$.

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

$NP(SP)$ contains the rest of the samples. We remark that a valuation s in $NP(SP)$ may or may not satisfy an invariant Inv which satisfies condition (1), (2) and (3).

Example 2. Take the program shown in Figure 1(a) as an example. Assume that the following three valuations are randomly generated: $(1, 2)$, $(10, 1)$ and $(100, 0)$ at line 1. Three sequences of valuations are generated after executing the program with these three valuations: $\langle(1, 2), (11, 5)\rangle$, $\langle(10, 1)\rangle$ and $\langle(100, 0)\rangle$ respectively. Note that the loop is skipped entirely for the latter two cases. After categorization, set $CE(SP)$ is empty; $Positive(SP)$ is $\{(1, 2), (11, 5)\}$; $Negative(SP)$ is $\{(100, 0)\}$; and $NP(SP)$ is $\{(10, 1)\}$.

After obtaining the samples and labeling them as discussed above, method $actL(SP)$ at line 4 in Algorithm 1 is invoked to generate a candidate invariant ϕ. We leave the details on how candidate invariants are generated in Section 3, which is our main contribution in this work. Once a candidate is identified, we move on to check whether ϕ satisfies condition (1), (2) and (3) at line 5. In particular, we check whether any of the following constraints is satisfiable or not using an SMT solver [4,15].

$$Pre \wedge \neg\phi \tag{4}$$
$$sp(\phi \wedge Cond, Body) \wedge \neg\phi \tag{5}$$
$$\phi \wedge \neg Cond \wedge \neg Post \tag{6}$$

Algorithm 2: Algorithm $actL(SP)$

```
1 while true do
2     if (CE(SP) is not empty)  exit and report "disproved";
3     let φ be a set of candidates generated by classify(SP);
4     if (φ is the same as last iteration) return φ;
5     add selectiveSampling(φ) into SP;
6     add all valuations s' such that s ⇒ s' for some s ∈ SP into SP;
```

where $sp(\phi \wedge Cond, Body)$ is the strongest postcondition obtained by symbolically executing program $Body$ starting from precondition $\phi \wedge Cond$ [16]. 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 active learning and so on.

Example 3. For the example shown in Figure 1(a), a candidate invariant which is automatically learned is $x - y \leq 16$. It is easy to check that this candidate satisfies all the three conditions and thus the Hoare triple shown in Figure 1(a) is proved. For Figure 1(c), a candidate invariant returned by method $actL(SP)$ is as follows.

$$490 + 16x - 9y \geq 0 \wedge 510 + 6x + 29y \geq 0 \wedge 56 - y \geq 0 \wedge 166 - 2x + 5y \geq 0$$

A counterexample $(-28, -11)$ is generated when we check the satisfiability of (4), which is then used to generate a new candidate. After multiple iterations of guess-and-check, the following invariant is generated.

$$1 + 2y \geq 0 \wedge 1 + 2x - 2y \geq 0 \wedge -1 + 2x \geq 0$$

Though different from the one we expect, this invariant turns out to be one which is strong enough to prove the Hoare triple.

3 Our Approach: Classification and Active Learning

In this section, we present details on how candidate loop invariants are generated. Algorithm 2 shows how $actL(SP)$ is implemented in general, i.e., it iteratively generates a candidate through classification (at line 3) and improves it through active learning (at line 5) until a fixed point is reached. Note that any time a counterexample is identified (at line 2), our approach exits and reports that the Hoare triple is disproved.

The method call $classify(SP)$ at line 3 in Algorithm 2 generates a candidate invariant based in classification. Intuitively, since we know that valuations in $Positive(SP)$ must satisfy Inv and valuations in $Negative(SP)$ must not satisfy Inv, a predicate

separating the two sets (a.k.a. a classifier) may be a candidate invariant. In the following, we fix two disjoint sets of samples P and N and discuss how to automatically generate classifiers separating P and N. For now, P can be understood as $Positive(SP)$ and N can be understood as $Negative(SP)$. We discuss alternatives in Section 3.4.

To automatically generate classifiers separating P and N, we apply existing classification techniques. There are many classification algorithms, e.g., perceptron [37], decision tree [40] and Support Vector Machine (SVM) [7]. In our approach, the classification algorithms must generate perfect classifiers. Formally, a perfect classifier ϕ for P and N is a predicate such that $s \in \phi$ for all $s \in P$ and $s \notin \phi$ for all $s \in N$. Furthermore, the classifier must be human-interpretable or can be handled by existing program verification techniques. In the following, we show how to adopt existing algorithms to generate candidate invariants based on SVM in Section 3.1. Next, we present an approach to generate disjunctive invariants in Section 3.2. We show how to improve all these candidates systematically through active learning in Section 3.3.

3.1 Linear and Polynomial Classifiers

In [49], the authors propose to use SVM to generate candidate invariants. SVM is a supervised machine learning algorithm for classification and regression analysis [7]. 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. We refer the readers to [39] for details on how the classifier is computed. In this work, we always choose the *optimal margin classifier* if possible. Intuitively, the optimal margin classifier could be seen as the strongest witness why P and N are different. 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 whereas c_i are constants.

In practice, linear classifiers may not be sufficient in proving certain Hoare triples and thus more expressive invariants are necessary. We can easily extend SVM to learn polynomial classifiers. Given P and N as well as a maximum degree d of the polynomial classifier, we can systematically map all the samples in P (similarly N) to a set of samples P' (similarly N') in a high dimensional space by expanding each sample with terms which have a degree up to d. For instance, assume that the maximum degree is 2, the sample valuation $\{x \mapsto 2, y \mapsto 1\}$ in P is mapped to $\{x \mapsto 2, y \mapsto 1, x^2 \mapsto 4, xy \mapsto 2, y^2 \mapsto 1\}$. SVM is then applied to learn a perfect linear classifier for P' and N'. Mathematically, a linear classifier in the high dimensional space is the same as a polynomial classifier in the original space [29]. We remark that the size of each sample in P' or N' grows rapidly with the increase of the degree and thus the above method is often limited to polynomial classifiers with relatively low degree.

A polynomial classifier can represent some classifiers in the form of disjunctive or conjunctive linear inequalities. For instance, the classifier $(x \geq d_0 \wedge x \leq d_1) \vee (x \geq d_2)$ where $d_0 < d_1 < d_2$ are constants can be represented equivalently as the following polynomial inequality.

$$x^3 + (d_0d_1 + d_0d_2 + d_1d_2)x^2 - (d_0 + d_1 + d_2)x - d_0d_1d_2 \geq 0$$

However, it is not always possible, i.e., some conjunctive or disjunctive linear inequalities cannot be expressed as a polynomial classifier. One example is: $x \geq 0 \wedge y \geq 0$.

In [49], an algorithm for learning conjunctive classifiers is proposed. The idea is to pick one sample s from N each time and identify a classifier ϕ_i (which could be a linear or polynomial one) to separate P and $\{s\}$, remove all samples from N which can be correctly classified by ϕ_i, and then repeat the process until N becomes empty. The conjunction of all the classifiers is then a perfect classifier separating P and N. We refer the readers to [49] for details of the algorithm. We remark that if we switch P and N, the negation of the learned classifier using this algorithm is a classifier which is in the form of a disjunction of linear inequalities.

3.2 Disjunctive Classifiers

It is often challenging to automatically generate disjunctive invariants [46,23], whereas certain Hoare triples can only be proved with disjunctive invariants. Two examples are shown in Figure 1(b) and Figure 1(d). In the following, we show how to learn disjunctive invariants through path-sensitive classification. Our observation is that disjunctive invariants are relevant often because the program contains branching commands (i.e., `if` and `while`). For instance, proving the Hoare triple shown in Figure 1(b) requires a disjunctive loop invariant, which is largely due to the conditional branch at line 3. Based on this observation, we apply the following approach to learn disjunctive invariants.

Assume that the loop body $Body$ contains a finite set of control locations. For instance, the loop in the first program in Figure 1(b) has four locations: line 3, 4, 5 and 6. Given a valuation of V, say s, we write $visit(s)$ to be the set of control location which is visited if we execute the program with variable valuation s during the first iteration of the loop. For instance, given the program in Figure 1(b), $visit(\{x \mapsto 0, y \mapsto -3\})$ returns $\{3, 5, 6\}$. If s does not satisfy the loop condition $Cond$, $visit(s)$ returns an empty set. We first partition P into a set of disjoint partitions such that all valuations in the same partition P_i visits the same set of control locations. For each partition P_i, we can construct the corresponding path condition pc_i.

Example 4. Given the program shown in Figure 1(b), if P is set to be $Positive(SP)$, we have three partitions. The first one contains all valuations s with $visits(s)$ being $\{3, 4\}$ whose path condition is $x + y \leq -2 \wedge x > 0$; the second one contains all valuations s with $visits(s)$ being $\{3, 5, 6\}$ whose path condition is $x + y \leq -2 \wedge x \leq 0$ and the last one contains all valuations s with $visits(s)$ being the empty set whose path condition is $x + y > -2$. The classifiers learned for these partitions depend on what samples we obtain.

Next, we apply the above-mentioned classification algorithms to learn a classifier for each partition, i.e., we learn a classifier ϕ_i for partition i separating P_i from N. Since any sample in P_i satisfies ϕ_i and pc_i (and every sample in N does not satisfies ϕ_i), the disjunction $\bigvee_i(\phi_i \wedge pc_i)$ is a perfect classifier separating P from N. Since ϕ_i could be a conjunctive predicate if we apply the algorithm in [49], we can learn candidate invariants in the form of disjunctions of conjunctions of linear or polynomial predicates.

Example 5. Though the program shown in Figure 1(d) contains no `if` command, variable valuations in $Positive(SP)$ can be partitioned into two: one containing those visit line 3 and 4, the other containing those skipping the loop. In the following, we show

how to learn a disjunctive loop invariant based on these two partitions. Note that a valuation s is in $Negative(SP)$ only if $s \in (y \leq 0 \wedge x \geq 0)$. If we have every valuation of V for these two partitions, a classifier we could learn for the former partition is $x < 0$ (i.e., a valuation must satisfy the invariant if it enters the loop) and the classifier we learn for the latter partition is $y > 0$. As a result, conjuncted with the path condition, we learn the candidate invariant: $(x < 0 \wedge x < 0) \vee (y > 0 \wedge x \geq 0)$, which is simplified as $x < 0 \vee y > 0$ and proves the Hoare triple.

We remark that in the above discussion, we assume that we can obtain every variable valuation, which is often infeasible in practice as there are too many of them. In the following subsection, we aim to solve this problem.

3.3 Active Learning

One fundamental problem on applying machine learning techniques to learn loop invariants is the we often have only a limited set of samples. That is, with the limited samples in $Positive(SP)$ and $Negative(SP)$, it is unlikely that we can obtain an "accurate" classifier. For instance, in Example 2, $Positive(SP)$ is $\{(1, 2), (11, 5)\}$ and $Negative(SP)$ is $\{(100, 0)\}$. A linear classifier identified using SVM for this example is: $3x - 10y \leq 152$. Although this classifier perfectly separates the two sets, it is not useful in proving the Hoare triple and is clearly the result of having limited samples. One obvious way to overcome this problem is to generate more samples. However, often a large number of samples are necessary in order to learn the correct classifier. One particular reason is that we often need the samples right on the classification boundary in order to learn the correct classifier, which are often difficult to obtain through random sampling. In existing guess-and-check approaches [49,48,47,45], the problem is overcome by checking whether the candidate invariant proves the Hoare triple through program verification. New samples are then provided as counterexamples by the program verification engine, which are used to refine the classifier. The issue of this approach is that often many iterations of guess-and-check are required so that the invariant would converge to the correct one.

Researchers in the machine learning community have studied extensively on how to overcome the problem of limited samples and one of the remedies is active learning [44]. Active learning is proposed in contrast to passive learning. A passive learner learns from a given set of samples which it has no control, whereas an active learner actively selects what samples to learn from. It has been shown that an active learner can sometimes achieve good performance using far less samples than would otherwise be required by a passive learner [50,51]. Active learning can be applied for classification or regression. In this work, we apply it for improving the candidate invariants generated by the above-discussed classification algorithms.

A number of different active learning strategies on how to select the samples have been proposed. For instance, version space partitioning [41] tries to select samples on which there is maximal disagreement between classifiers in the current version space (e.g., the space of all classifiers which are consistent with the given samples); uncertainty sampling [33] maintains an explicit model of uncertainty and selects the sample that it is least confident about. The effectiveness of these strategies can be measured in terms of the labeling cost, i.e., the number of labeled samples needed in order to learn a

classifier which has a classification error bounded by some threshold ϵ. For some classification algorithms, it has been shown that active learning reduces the labeling cost from $\Omega(\frac{1}{\epsilon})$ to the optimal $O(d \lg \frac{1}{\epsilon})$ where d is the dimension of samples [20,14]. That is, if passive learning requires a million samples, active learning may require just $\lg 1000000$ (≈ 20) to achieve the same accuracy.

In this work, we adopt the active learning strategy for SVM proposed in [42], called selective sampling, to improve the invariant candidates. This strategy has been shown to be effective in achieving a high accuracy with fewer examples in different applications [50,51]. In particular, at line 5 of Algorithm 2, after obtaining a classifier ϕ based on existing samples in SP, we apply method $selectiveSampling(\phi)$ to selectively generate new samples. It works by generating multiple samples on the current classification boundary ϕ. Afterwards, the samples are added into SP at line 5 and 6 and we repeat from line 2 until the classifier converges.

For classifiers in the form of linear inequalities, identifying samples on the classification boundary is straightforward. In the above example, given the current classifier $3x - 10y \leq 152$, we apply selective sampling and generate new valuations $(7, 13)$ and $(14, -11)$ by solving the equation $3x - 10y = 152$. For classifiers in the form of polynomial inequalities, the problem is more complicated since existing solvers for multi-variable polynomial equations have limited scalability. We thus use a simple approach to identify solutions of a polynomial equation, which we illustrate through an example in the following. Assume that we learn the classifier: $-4x^2 + 2y \geq -11$. The following steps are applied for selective sampling.

1. Choose a variable in the classifier, e.g., x.
2. Generates random value for all other variables. For example, we let y be 12.
3. Substitute the variables in the classifiers with the generated values and solve the univariable equation, e.g., $-4x^2 + 24 = -11$. If there is no solution, go back to (1) and retry. In our example, $x \approx 2.9580$.
4. Roundoff the values of all the variables according to their types in the program. In our example, we obtain the valuation $(3, 12)$.

In the case that a conjunctive or disjunctive classifier is learned, we apply the above selective sampling approach to each and every clause in the classifier to obtain new samples. With the help of active learning and selective sampling, we can often reduce the number of learn-and-check iterations. As we show in Section 4, often one iteration of guess-and-check is sufficient to prove the Hoare triple.

Selective sampling vs. other sampling In the following, we briefly discuss why selective sampling is helpful from a high-level point of view. In this work, we collect samples in three different ways. Firstly, random sampling provides us an initial set of samples. The cost of generating a random sample is often low. However we often need a huge number of random samples in order to learn accurately. Secondly, selective sampling has a slightly higher cost as it requires to solve some equation system. However, it has been shown that selective sampling is often beneficial compared to random sampling [50,51]. The last way of sampling is sampling through verification. When a candidate invariant fails any of the three conditions (1), (2) and (3) in the candidate verification stage, the verifier provides counter-examples, which are added as new samples. Sampling through

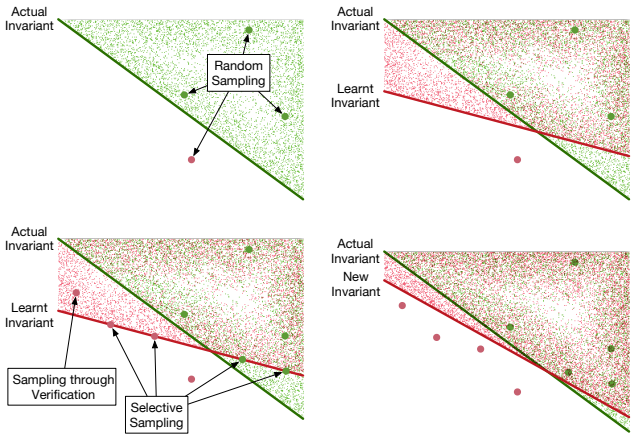

Fig. 2: Sampling approaches

verification provides useful new samples by paying a high cost. Thus, in this work, our approach is to start with random sampling, use selective sampling to improve the classifier as much as possible and apply sampling through verification only as the last resort.

Figure 2 visualizes how different sampling methods work in a 2-D plane. We start with the figure in the top-left corner, where the dots are the samples obtained through random sampling. The (green) area above the line represents the space covered by the actual invariant. Based on these samples, a classifier (shown as the red line) is learnt to separate the random samples, as shown in the top-right figure. Selective sampling allows us to identify those samples on the classification boundary based on the learnt classifier, as shown in the bottom-left figure. In comparison, sampling through verification would provide us a sample between the two lines, as shown in the bottom-left figure. The classifier will be improved by either selective sampling or sampling through verification, as shown in the bottom-right figure. The benefit of always applying selective sampling before applying sampling through verification is that verification is often costly. Even worse, it may not be available sometimes due to the limitation of existing program verification techniques. Thus we would like to avoid it as much as possible.

3.4 Making Use of Undetermined Samples

So far we have focused on learning and refining classifiers between $Positive(SP)$ and $Negative(SP)$ as candidate invariants. The question is then: how do we handle those valuations in $NP(SP)$? If we simply ignore them, there may be a gap between $Positive(SP)$ and $Negative(SP)$ and as a result, the learnt classifier may not converge to the invariant we want, even with the help of active learning. This is illustrated in Figure 3, where the set of valuations in $Positive(SP)$ (marked with $+$), $Negative(SP)$ (marked with $-$) and $NP(SP)$ (marked with ?) for the example in Figure 1(a) are visualized in a 2-D plane. Many samples between the line $x = y$ and $x - y = 16$ may be contained in $NP(SP)$. As a result, without considering the sam-

ples in $NP(SP)$, a classifier located in the $NP(SP)$ region (e.g., $x - y \leq 10$, or $x - y \leq 13$) may be learned to perfectly classify $Positive(SP)$ and $Negative(SP)$. Worse, identifying more samples may not be helpful in improving the classifier if the new samples are in $NP(SP)$.

To overcome the problem, in addition to learn a classifier separating $Positive(SP)$ and $Negative(SP)$, we learn candidate invariants making use of $NP(SP)$. That is, we learn classifiers separating $Positive(SP)$ from $Negative(SP) \cup NP(SP)$ (i.e., assuming valuations in $NP(SP)$ fail the actual invariant), and classifiers separating $Negative(SP)$ from $Positive(SP) \cup NP(SP)$ (i.e., assuming valuations in NP satisfy the actual invariant). For the example in Figure 1(a), if we focus classifiers in the form of linear inequalities, the classifier separating $Positive(SP)$ from the rest converges to NULL (no such classifier), whereas the classifier separating $Negative(SP)$ from the rest converges to $x - y \leq 16$, which can be used to prove the Hoare triple. Note that this is orthogonal to which classification algorithm is used and whether selective sampling is applied.

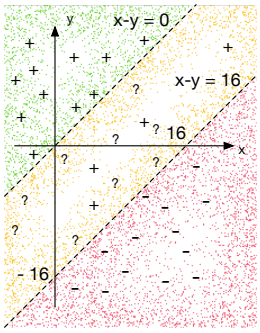

Fig. 3: Samples visualization

4 Evaluation

We have implemented our approach for loop invariant generation in a tool called ZILU (available at [1]). ZILU is written using a combination of C++ and shell codes (for invoking external tools). It makes use of GSL [21] to solve equation systems; and uses LibSVM [9] for SVM-based classification. For candidate invariant verification, we modify the KLEE project [8] to symbolically execute C programs prior to invoking Z3 [15] for checking satisfiability of condition (4), (5) and (6). We remark that KLEE is a concolic testing engine and thus it may concretely execute the programs and return under-approximated abstraction. This may affect the soundness of our system. To overcome this problem, we detect those path conditions produced from concrete executions and return sound abstraction (i.e., $true$).

Our evaluation subjects include a set of more than 50 C programs we can find from previous publications (e.g., [23,49,22,30,17]) as well as software verification competitions [5] (excluding those which cannot be made to satisfy our assumptions). All evaluated programs are available at [1]. We remark that the loops in these benchmark programs often contain non-deterministic choices which are used to model I/O environment (e.g., an external function call). As non-determinism is beyond the scope of this work, we replace these non-deterministic commands with free boolean variables. The parameters in our experiments are set as follows. For random sampling, we generate 8 random values for every input variables of a program from their default ranges. Furthermore, during selective sampling, in addition to samples nearby the classification boundary, we add a few random samples in order to improve convergence [42]. The ratio between random samples and selective samples is 1:4. When we invoke LibSVM

for classification, the parameter C (which controls the trade-off between avoiding misclassifying training examples and enlarging decision boundary) and the inner iteration for SVM learning are set to their maximum value so that it generates only perfect classifiers. During candidate verification, integer-type variables in programs are encoded as integers in Z3 (not as bit vectors). Since we have different ways of setting the samples for classification, e.g., by setting the two sets of samples P and N differently as discussed in Section 3.4, and different classification algorithms (linear vs. polynomial or conjunctive vs. disjunctive), we simultaneously try all combinations and terminate as soon as either the Hoare triple is proved or disproved. For polynomial classifiers, the maximum degree is bounded by 4. Furthermore, we look for a polynomial classifier with degree d only if we cannot find any polynomial classifier with lower degree.

All of the experiments are conducted using x64 Ubuntu 14.04.1 (kernel 3.19.0-59-generic) with 3.60 GHz Intel Core i7 and 32G DDR3, where to means time out after 6 minutes. Each experiment is executed three times since there is randomness in our approach and we report the average as the result. Due to space limit, we present the results on 35 programs in Table 1. These programs are selected as there are less variance in the experiment results across different runs of the same program (due to randomness in our approach). We refer the readers to [1] for the full details. The first column shows the name of the benchmark program as well as where it is from. Note that we created a few programs which require polynomial or disjunctive loop invariant due to the lack of such examples in existing work. The second column shows the type of invariant required for proving the Hoare triple. The next three columns present statistics of ZILU with selective sampling, i.e., the number of samples generated in total, the number of guess-and-check iterations and the total execution time. In order to show the relevance of active learning and selective sampling, we measure the performance of ZILU [1] without selective sampling as well. The next three columns present the corresponding statistics. The winner of each measurement is highlighted using a bold font. *to* denotes timeout (360seconds). The second last column shows the result of an existing tool called Interproc, i.e., whether it generates a correct invariant. We do not show the time of Interproc because its result may not be correct since it does not prove/disprove the Hoare triples. Interproc generates invariants based on abstract interpretation. In the experiments, it is set to use its most expressive abstract domain, i.e., the reduced product of polyhedra and linear congruences abstraction. The last column shows the verification result of a state-of-the-art program verifier CPAChecker [6] where ✗ means that CPAChecker produces a false positive as the verification result. The binary we use is the version use for SV-COMP 2016 [5]. Note that the comparison between ZILU and Interproc or CPAChecker should be taken with a grain of salt as the methods are different.

We have the following observations based on the experiment results. First, ZILU is effective in proving these programs. For all of these programs, ZILU is able to find a loop invariant which proves the Hoare triple. In comparison, Interproc failed in 13 cases and CPAChecker failed in 18 cases (with 8 timeouts and 10 false positives). Secondly, ZILU is relatively efficient. For all these programs, ZILU finishes the prove within 75 seconds. A close look reveals that most of the time is spent on classification and selective sampling. It should be noted though that when CPAChecker is able to prove the Hoare triple, it is usually very efficient. Thirdly, selective sampling is helpful in reduc-

Table 1: Experiment results

		ZILU + Selective Sampling			ZILU - Selective Sampling				
benchmark	inv type	♯sample	♯iteration	time(s)	♯sample	♯iteration	time(s)	Interproc	CPAChecker
05 [17]	linear	107	**1**	**4**	**100**	2	**4**	✓	to
21 [17]	linear	**133**	**1**	**4**	to	to	to	✗	✗
23 [17]	conjunctive	**255**	**3**	**5**	283	8	9	✗	to
28 [17]	conjunctive	307	**4**	8	**233**	**4**	**7**	✓	to
30 [17]	conjunctive	**1565**	**43**	**26**	to	to	to	✗	4
35 [17]	linear	133	**1**	**4**	**120**	2	**4**	✓	2
43 [17]	linear	**300**	**1**	6	323	3	**5**	✓	2
bound [17]	linear	73	**1**	**4**	**70**	2	**4**	✓	3
down [26]	linear	**300**	**4**	**5**	350	**4**	6	✓	to
f2 [1]	linear	**100**	**1**	**4**	140	2	4	✓	✗
fig1b	disjunctive	**3533**	**7**	**34**	6400	10	51	✗	1
fm11 [43]	conjunctive	**166**	**2**	**12**	to	to	to	✓	1
interproc1 [30]	linear	**293**	**5**	**6**	to	to	to	✓	2
interproc2 [30]	linear	133	**1**	**7**	**93**	2	9	✓	2
interproc3 [30]	linear	**180**	**1**	8	230	3	**6**	✓	✗
interproc4 [30]	linear	**80**	**1**	**6**	120	2	**6**	✗	✗
multivar_1 [30]	conjunctive	233	**2**	6	**200**	3	**5**	✓	2
pldi08_fig1 [22]	disjunctive	**583**	**2**	**7**	777	**2**	13	✗	✗
pldi08_fig7 [22]	linear	**27**	**1**	**4**	43	2	**4**	✓	2
terminator_01 [5]	linear	53	**1**	**4**	**50**	2	4	✓	2
up_true_2 [5]	conjunctive	920	33	31	**600**	**8**	**20**	✓	to
xle10 [49]	linear	**57**	**1**	**4**	60	2	4	✓	2
xy0_1 [49]	conjunctive	273	**4**	7	**220**	4	**6**	✓	to
xy0_2 [49]	conjunctive	**193**	**3**	**6**	**187**	4	**6**	✓	to
xy4_1 [49]	conjunctive	**220**	**3**	**6**	313	5	8	✓	to
xyle0 [49]	polynomial	433	**4**	91	**267**	5	**79**	✗	2
xyz_2 [49]	conjunctive	**470**	**5**	**6**	to	to	to	✓	✗
zilu_conj1	conjunctive	**180**	**1**	**32**	to	to	to	✗	2
zilu_disj1	disjunctive	**2740**	**4**	**46**	4160	**4**	65	✓	✗
zilu_disj2	disjunctive	**3193**	**3**	**27**	5887	**3**	30	✗	✗
zilu_disj3	disjunctive	**7382**	**3**	**29**	8050	4	40	✗	✗
zilu_poly1	polynomial	**57**	**2**	**37**	117	5	80	✗	2
zilu_poly3	polynomial	50	**1**	**7**	**43**	2	9	✗	2
zilu_poly6	polynomial	**240**	**4**	**75**	to	to	to	✗	✗

ing the number of samples and guess-and-check iterations. In all but one experiments, ZILU is able to generate the invariant with fewer or equal number of guess-and-check iterations if selective sampling is applied. Though it rarely happens, due to the randomness in our approach, it may happen that the right invariant is learned by luck with few samples. This is happened in the one case where ZILU without selective sampling has less iterations. It should be noted that for 7 programs, ZILU is unable to learn the invariant without the help of selective sampling, i.e., it timeouts due to too many guess-and-check iterations. *We would like to highlight that for 14 programs,* ZILU *is able to learn the correct invariant with one guess-and-check iteration with selective sampling*, whereas this is never the case without selective sampling. Furthermore, it happens more often when the invariant is a linear predicate. We remark that being able to learn the correct invariant without program verification is useful for handling complex programs. That is, even if we are unable to automatically verify the generated invariant due to the limitation of existing program verification techniques, ZILU's result is still useful in these cases as the generated invariant can be used to manually verify the program.

Lastly, ZILU often takes more samples or guess-and-check iterations to learn conjunctive or disjunction invariants. For conjunctive invariants, this is because the algorithm [49] for learning conjunctive classifiers often requires more samples before con-

vergence. For disjunction invariants, this is because we need sufficient samples in each partition in order to learn the right invariant. In terms of time, ZILU often takes more time to learn conjunctive, polynomial or disjunctive invariants. This is because in such a case, SVM classification is invoked many times in one guess-and-check iteration.

5 Conclusion and Related Work

In this work, we propose an approach to improve loop invariant generation through guess-and-check. In particular, we propose to apply active learning techniques so as to learn accurate candidate loop invariants prior to the invariant checking phase. Furthermore, we propose a path-sensitive way of learning disjunctive loop invariants through classification. In principle, our approach can be extended to learn arbitrary mathematical classifiers using methods like SVM with kernel methods [29]. Nonetheless, we focus on invariants in the form of polynomial inequalities or conjunctions/disjunctions of polynomial inequalities in our evaluation.

This work is closely related and inspired by the guess-and-check approach for loop invariant generation, documented in [49,48,47,45]. In [49], the authors proposed to learn loop invariants based on SVM classification. The samples are generated through constraint solving. In [48], the authors proposed to apply PAC learning. It has been demonstrated that their approach may learn invariants in the form of arbitrary boolean combinations of polynomial inequalities under certain assumptions. In [47], the authors developed a guess-and-check algorithm for generating algebraic equation invariants. In [45], the authors proposed a framework for generating invariant based on randomized search. In particular, their approach has two phases. The search phase uses randomized search to discover candidate invariants and the validate phase uses the checker to either prove or refute the candidate. ZILU complements the above approaches with active learning so as to reduce the need of checking, sometimes completely. Furthermore, ZILU supports a new way of learning disjunctive invariants

In addition, this work is related to a large body of approaches on loop invariant generation. The existing approaches can be mainly categorized as: the ones based abstract interpretation [13,36,31,32], the ones based on constraint synthesis [25,12,24], the ones based on counterexample-guided abstraction refinement ($CEGAR$) [28,3,11], the ones based on computing interpolation [27,34,35], the ones based on abductive inference [17], the ones based on guess-and-check [19,18].

On one hand, invariant inference methods based on abstract interpretation and constraint synthesis often try to generate all possible invariants in certain domain [36,32,25], regardless of whether they are useful to prove the Hoare triple or not. As a result, the invariants inferred by them can be complex sometimes and yet fail to prove the program correctness. On the other hand, other methods based on $CEGAR$, interpolation and abduction only generate those related to the program verification [17]. Different from the above approaches, ZILU initially treats the given program as a black box and only collects control flow information and relevant program states by executing the program. This step has no scalability issue. ZILU only opens up the black box after candidate invariants have converged. From this point of view, ZILU is lightweight compared to the above approaches.

References

1. ZILU repo. `https://github.com/lijiaying/ZILU`, 2016.
2. D. Babic, B. Cook, A. J. Hu, and Z. Rakamaric. Proving termination of nonlinear command sequences. *Formal Asp. Comput.*, 25(3):389–403, 2013.
3. T. Ball and S. K. Rajamani. The SLAM toolkit. In *CAV*, pages 260–264, 2001.
4. C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.
5. D. Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In *TACAS*, pages 887–904, 2016.
6. D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *CAV*, pages 184–190, 2011.
7. B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *workshop on Computational learning theory*, pages 144–152. ACM, 1992.
8. C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
9. C.-C. Chang and C.-J. Lin. Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.
10. H. Chen, C. David, D. Kroening, P. Schrammel, and B. Wachter. Synthesising interprocedural bit-precise termination proofs (T). In *ASE*, pages 53–64, 2015.
11. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
12. M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, pages 420–432, 2003.
13. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM, 1978.
14. S. Dasgupta. Coarse sample complexity bounds for active learning. In *NIPS*, pages 235–242, 2005.
15. L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
16. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.
17. I. Dillig, T. Dillig, B. Li, and K. L. McMillan. Inductive invariant generation via abductive inference. In *OOPSLA*, pages 443–456, 2013.
18. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *SCP*, 69(1):35–45, 2007.
19. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *Formal Methods Europe*, pages 500–517, 2001.
20. R. Gilad-Bachrach, A. Navot, and N. Tishby. Query by committee made real. In *NIPS*, pages 443–450, 2005.
21. B. Gough. *GNU scientific library reference manual*. Network Theory Ltd., 2009.
22. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *TACAS*, pages 443–458. Springer, 2008.
23. S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292, 2008.
24. S. Gulwani, S. Srivastava, and R. Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI*, pages 120–135, 2009.
25. A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In *Proceedings of 21st International Conference on Computer Aided Verification*, pages 634–640, 2009.

26. A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In *International Conference on Computer Aided Verification*, pages 634–640. Springer, 2009.
27. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
28. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Model Checking Software*, pages 235–239. Springer, 2003.
29. T.-M. Huang, V. Kecman, and I. Kopriva. *Kernel based algorithms for mining huge data sets*, volume 1. Springer, 2006.
30. B. Jeannet. Interproc analyzer for recursive programs with numerical variables. *http://pop-art.inrialpes.fr/interproc/interprocweb.cgi*, pages 06–11, 2010.
31. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976.
32. V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *VMCAI*, pages 229–244, 2009.
33. D. D. Lewis and W. A. Gale. A sequential algorithm for training text classifiers. In *SIGIR Forum*, pages 3–12, 1994.
34. K. L. McMillan. Interpolation and sat-based model checking. In *Computer Aided Verification*, pages 1–13, 2003.
35. K. L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification*, pages 123–136, 2006.
36. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
37. M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry, 2nd edition*. The MIT Press, 1972.
38. C. S. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, Oct. 2009.
39. J. Platt et al. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.
40. J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.
41. R. A. Ruff and T. G. Dietterich. What good are experiments? In *Proceedings of the Sixth International Workshop on Machine Learning (ML 1989)*, pages 109–112, 1989.
42. G. Schohn and D. Cohn. Less is more: Active learning with support vector machines. In *ICML*, pages 839–846, 2000.
43. D. Schwartz-Narbonne, P. Rümmer, M. Schäf, A. Tiwari, and T. Wies. Non-monotonic program analysis.
44. B. Settles. *Active Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.
45. R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *Computer Aided Verification*, pages 88–105. Springer, 2014.
46. R. Sharma, I. Dillig, T. Dillig, and A. Aiken. Simplifying loop invariant generation using splitter predicates. In *CAV*, pages 703–719, 2011.
47. R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, pages 574–592, 2013.
48. R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *Static Analysis Symposium*, pages 388–411, 2013.
49. R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *Computer Aided Verification*, pages 71–87. Springer, 2012.
50. S. Tong and E. Y. Chang. Support vector machine active learning for image retrieval. In *Proceedings of the 9th ACM International Conference on Multimedia*, pages 107–118, 2001.
51. S. Tong and D. Koller. Support vector machine active learning with applications to text classification. *Journal of Machine Learning Research*, 2:45–66, 2001.