Automatic Loop-invariant Generation and Refinement through Selective Sampling

Jiaying Li[1], Jun Sun[1], Li Li[1], Quang Loc Le[2] and Shang-Wei Lin[3]
[1]Singapore University of Technology and Design, Singapore
[2]School of Computing, Teesside University, United Kingdom
[2]School of Computer Science and Engineering, Nanyang Technological University, Singapore

***Abstract*—Automatic loop-invariant generation is important in program analysis and verification. In this paper, we propose to generate loop-invariants automatically through learning and verification. Given a Hoare triple of a program containing a loop, we start with randomly testing the program, 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 a candidate loop-invariant automatically. Afterwards, we refine the candidate through selective sampling so as to overcome the lack of sufficient test cases. Only after a candidate invariant cannot be improved further through selective sampling, we verify whether it can be used to prove the Hoare triple. If it cannot, the generated counterexamples are added as new tests and we repeat the above process. Furthermore, we show that by introducing a 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. In order to evaluate our idea, a prototype tool has been developed and the experiment results show that our approach complements existing approaches.**

***Index Terms*—loop-invariant, program verification, classification, active learning, selective sampling**

I. 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, review existing approaches and then briefly introduce our proposal. Without loss of generality, 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 that updates the valuation of V. For simplicity, we assume that it is a deterministic function[1] 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 goal is to either prove or disprove the Hoare triple. To prove it, we would like to find a loop-invariant Inv which satisfies the following three conditions.

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

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

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

To disprove it, we would like to find a valuation s such that $s \in Pre$ and executing the loop until it terminates results in a valuation s' such that $s' \notin Post$. For simplicity, we assume that the loop always terminates and refer the readers to [2], [10] for research on proving loop termination.

Loop-invariant generation is a long standing problem. Many approaches have been proposed to solve this problem [13], [36], [31], [28], [3], [11], [27], [34], [35], [25], [12], [24], [17]. These approaches all rely on some form of constraint solving and often suffer from scalability issues. Recently, a number of guess-and-check approaches [48], [47], [46], [44], [19], [18] have been proposed. These approaches start with generating a set of valuations of V (a.k.a. the samples) and categorize them into different groups, e.g., one containing those satisfying the loop-invariant and another containing those not. Learning techniques are then applied to generalize the valuations in a certain form to *guess* candidate loop-invariants. The candidates are then *check*ed using program verification techniques (like symbolic execution [38]) to see whether they satisfy the three conditions. If any of the conditions is violated, counterexamples in the form of variable valuations can be obtained. For instance, given a candidate loop-invariant 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 this sample s, we can learn a new candidate invariant. This guess-and-check process is repeated until the Hoare triple is either proved or disproved.

Existing guess-and-check approaches vary in how samples are generated and candidate invariants are guessed. We refer the readers to Section V for a detailed discussion. A common problem with these guess-and-check approaches is that their effectiveness is often limited by the samples generated in their first phases. In order to guess the right invariant, often a large

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

number of samples are necessary. If classification techniques are employed, 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 however 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 [48], [47], [46], [44] are often limited, e.g., conjunctive linear inequalities [48] or equalities [46]. Despite the approaches presented in [23], [45], learning disjunctive loop-invariants remains a challenge.

Our Contribution In this work, we propose a technique to improve the existing guess-and-check approaches [48], [47], [46], [44] by making the following contributions. Firstly, we propose an active learning technique, known as selective sampling, to overcome the limitation of random sampling. That is, selective sampling 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 using heavy program verification techniques. As a result, we can reduce the number of guess-and-check iterations. Secondly, we propose to generate disjunctive invariants through *path-sensitive* learning. That is, we partition the samples according to the control locations they visit, classify each partition separately and construct a disjunction of the learned results for each partition as the loop-invariant. Thirdly, our approach is designed to be extensible so that we can learn different kinds of invariants. For instance, we generate candidate invariants in the form of polynomial inequalities or their conjunctions when different classification algorithms are adapted. 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], CPAChecker [7], InvGen [26] and BLAST [6]. Most test subjects are gathered from previous collections as well as the software verification repository [5]. The results show that, for those programs that are compatible with ZILU's input restrictions, ZILU is able to prove the maximum number of programs. Furthermore, it is shown that selective sampling is able to reduce the need for checking, sometimes completely.

Organization The remainders of the paper are organized as follows. Section II presents an overview of our approach using simple illustrative examples. Section III shows how candidate loop-invariants are generated through classification and refined through selective sampling. Section IV evaluates our approach using a set of benchmark programs. Section V reviews related work and Section VI concludes.

II. THE OVERALL APPROACH

Through this paper, 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. Four Hoare triple examples are shown in Figure 1, where an `assume` statement captures the precondition and an `assert` statement captures the postcondition. The set V for each program contains two integer variables: x and y. For simplicity, we write (a, b) where a and b are integer constants to denote the evaluation $\{x \mapsto a, y \mapsto b\}$. Furthermore, 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 rest of the Hoare triples. We remark that there might be different loop-invariants which could be used to prove the Hoare triples. In the following, we show how we generate loop-invariants for proving these Hoare triples.

Our overall approach is shown in Algorithm 1. We start with randomly generating a set of valuations of V, denoted as SP, at line 1 (a.k.a. random sampling). Random sampling provides us an initial set of samples to learn the 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 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 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. Formally,

$$\begin{aligned} CE(SP) = \{ & s \in SP \mid \exists s_0, s'. \\ & s_0 \in Pre \wedge s_0 \Rightarrow s \Rightarrow s' \\ & \wedge s' \notin Cond \wedge s' \notin Post \} \end{aligned}$$

A valuation s in $CE(SP)$ starts from a valuation s_0 which satisfies Pre and becomes a valuation s' which fails $Post$

```
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

when the loop terminates. If $CE(SP)$ is non-empty, the Hoare triple is disproved.

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

$Positive(SP)$ 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 s', which satisfies $Post$ when the loop terminates. Let Inv be any loop-invariant that 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 $s \in Inv$.

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

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

Algorithm 1: Algorithm $verify()$

```
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 activeLearn(SP) to generate a candidate
      invariant φ;
5     return "proved" if the program is verified with φ
      otherwise add the counterexample into SP;
```

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

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

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

where $sp(\phi \wedge Cond, Body)$ is the strongest postcondition obtained by symbolically executing program $Body$ starting from precondition $\phi \wedge Cond$ [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

Algorithm 2: Algorithm $activeLearn(SP)$

```
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;
```

$activeLearn(SP)$ 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 (5), which is then used to generate a new candidate. After multiple iterations of guess-and-check, the following invariant is generated.

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

Different from the invariant in Figure 1(d), this candiate still succeeds in proving the given Hoare triple. Thus, the loop-invariant is found.

III. Our Approach: Classification, Active Learning and Selective Sampling

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

The method call $classify(SP)$ in Algorithm 2 generates a candidate invariant based on classification techniques. 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 III-D.

To automatically generate classifiers separating P and N, we apply existing classification techniques. There are many classification algorithms, e.g., [37], [40], [8]. 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 first briefly discuss how to generate conjunctive invariants using the approach proposed in [48] and then propose a path-sensitive approach to generate disjunctive invariants. Afterwards, we show how to improve candidate invariants systematically through selective sampling.

A. Conjunctive Invariants

In the following, we show how to generate loop-invariants in the form: $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_k$ where each ϕ_i is a polynomial inequality up to certain degree, constituted by variables in V. Our approach is based on Support Vector Machines (*SVM*).

SVM is a supervised machine learning algorithm for classification and regression analysis [8]. In general, the binary classification functionality of *SVM* works as follows. Given P and N, *SVM* can generate 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 and c_i are constant coefficients.

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]. Note that the size of each sample in P' or N' grows rapidly with the increase of the degree and thus the above method is limited to polynomial classifiers with a 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, this representation transformation is not always possible, i.e., some conjunctive or disjunctive linear inequalities cannot be expressed as a polynomial classifier. One typical example is: $x \geq 0 \wedge y \geq 0$.

To generate conjunctive classifiers, we adopt the algorithm *SVM-I* proposed in [48]. The idea is to pick one sample s from N each time and identify a classifier ϕ_i in the form of a polynomial inequality 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 ϕ_i is then a perfect classifier separating P and N. We refer the readers to [48] for details of the algorithm. We remark that if we switch P and N, the

negation of the learned classifier using this algorithm is in the form of a disjunction of polynomial inequalities.

B. Disjunctive Invariants

It is often challenging to automatically generate disjunctive invariants [45], [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 one way to learn disjunctive invariants, i.e., invariants in the general form of

$$\phi_1 \vee \phi_2 \vee \cdots \vee \phi_m$$

where each $\phi_i = \varphi_{i,1} \wedge \varphi_{i,2} \wedge \cdots \wedge \varphi_{i,n}$ is a conjunctive polynomial inequality. Our observation is that disjunctive invariants are often required to prove certain Hoare tripe 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 branch at line 3. Based on this observation, we propose to learn disjunctive invariants through path-sensitive classification.

Without loss of generality, we assume that the loop body $Body$ can be modeled as a transition system $(C, init, end, T, L)$. C is a finite set of control locations. $init \in C$ is a unique entry point (i.e., the start of the program). $end \in C$ is a unique exit point (i.e., the end of the program, which is assumed to be always reachable). $T : C \to C$ is a transition function[2] which captures the control flow. Lastly, L is a labeling function which labels each transition with a pair (g, f) where g is a guard condition and f is a function updating variable valuation. Note that g is used to model branching conditions whereas f is used to model program statements like assignments. For instance, the loop body in the first program in Figure 1(b) can be modeled as a transition system with four control locations representing line 3, 4, 5 and 6; and the transition from the control location representing line 3 to the one representing line 4 is labeled with a guard condition $x > 0$ and a function which does not change any variable valuation.

Given a valuation s of V satisfying the loop condition $Cond$, we can obtain a unique path through the program $path(s) = \langle c_1, c_2, \cdots, c_k \rangle$ where $c_i \in C$ for all i such that $c_1 = init$, $c_k = end$ and every guard condition along the path is satisfied. For instance, given the loop body in Figure 1(b) and valuation $\{x \mapsto 0, y \mapsto -3\}$, the unique path is $\langle 3, 5, 6 \rangle$. If s violates the loop condition $Cond$, we set $path(s)$ to be an empty sequence. Intuitively, $path(s)$ is the set of sequence of control locations visited by s in one iteration of the loop.

Our path-sensitive classification starts with partitioning P into a set of disjoint partitions such that for each partition P_i, $path(s) = path(s')$ for all valuation s and s' in P_i. For each P_i, we can construct a unique path condition pc_i, i.e., a formula over the symbolic variables in V and the accumulated constraints which the symbolic variables must satisfy in order for an execution to follow the corresponding path. For instance, 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 $path(s)$ being $\langle 3, 4 \rangle$ whose path condition is $x + y \leq -2 \wedge x > 0$; the second one contains all valuations s with $path(s)$ being $\langle 3, 5, 6 \rangle$ whose path condition is $x + y \leq -2 \wedge x \leq 0$ and the last one contains all valuations s with $path(s)$ being $\langle \rangle$ whose path condition is $x + y > -2$.

Next, we apply the approach presented in Section III-A to learn a conjunctive classifier for each partition P_i, i.e., we learn a classifier ϕ_i for separating P_i from N_i. Then the disjunction $\bigvee_i (\phi_i \wedge pc_i)$ is a perfect classifier separating P from N. Since ϕ_i is a conjunctive predicate, we learn candidate invariants in the form of disjunction of conjunction of polynomial inequalities.

Example 4. Though the program shown in Figure 1(d) contains no `if` command, variable valuations in $Positive(SP)$ can be partitioned into two partitions according to our definition: 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 can be 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.

C. Active Learning and Selective Sampling

One fundamental problem with applying machine learning techniques to learn loop-invariants is that 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, as shown 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 [48], [47], [46], [44], [19], [18], the problem is overcome by checking whether the candidate invariant proves the Hoare triple through program verification. That is, new samples are obtained from counterexamples generated by the program verification engine,

[2]It is a function as we assume $Body$ is deterministic.

which are then used to refine the classifier. The issue is that often many iterations of guess-and-check are required before 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. One of the remedies is active learning [43]. Active learning is proposed in contrast to passive learning. A passive learner learns from a given set of samples that it has no control over, 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 fewer samples than would otherwise be required by a passive learner [49], [50]. 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 [32] 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 the samples [21], [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 [49], [50]. 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.

The implementation of $selectiveSampling$ depends on the type of classifiers. For classifiers in the form of linear inequalities, identifying samples on the classification boundary is straightforward, i.e., by solving an equation. 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 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 the empirical studies shown in Section IV, one iteration of guess-and-check is sufficient in some cases to prove the Hoare triple.

Advantages of Selective 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 us to solve some equation system. However, it has been shown that selective sampling is often beneficial compared to random sampling [49], [50]. 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 verification provides useful new samples by paying a high cost. Furthermore, for complex programs, sampling through verification may not be feasible due to the limited capability of existing program verification techniques. 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 along the classification boundary, 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

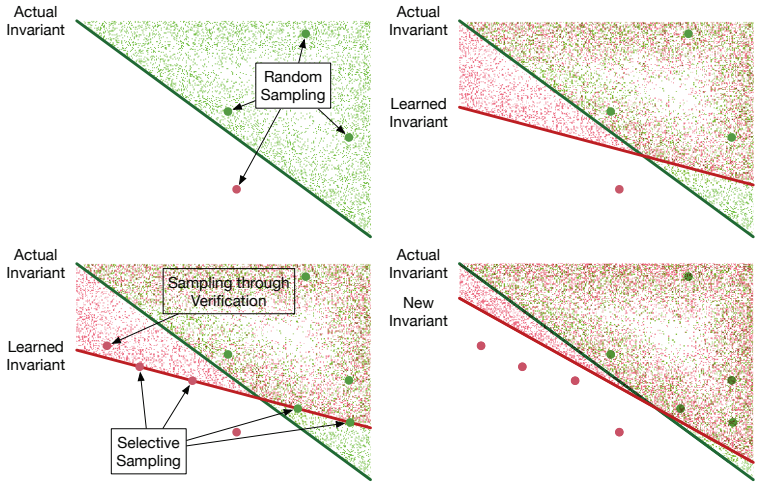

Fig. 2: Sampling approaches

always applying selective sampling before applying sampling through verification is that verification is often costly or even worse, not available due to the limitation of existing program verification techniques. Thus we would like to avoid it as much as possible.

D. 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 samples 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 solve the problem, in addition to learn a classifier separating $Positive(SP)$ and $Negative(SP)$, we learn candidate invariants making use of $NP(SP)$. In principle, we should enumerate all the possible categorization of the samples in $NP(SP)$ and run classification algorithm on each of them. However at most time it is very time-consuming and instead we only try two extreme case in our implementation, which is far from perfect and will be refined in the future. In our current setting, 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

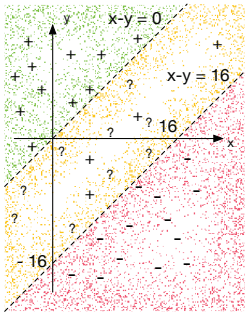

Fig. 3: Samples visualization

in Figure 1(a), if we focus on 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.

IV. Implementation and Evaluation

We have implemented our approach for loop-invariant generation in a tool called ZILU (available at [1]). For candidate-invariant verification, we modify the KLEE project [9] to symbolically execute C programs prior to invoking Z3 [15] for checking satisfiability of condition (4), (5) and (6). We remark that, as a concolic testing engine, KLEE 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 a sound abstraction (i.e., $true$).

Our evaluation subjects include a set of C programs gathered from multiple resources, such as previous publications (e.g., [23], [18], [48], [22], [30], [17]) and the software verification competitions 2017 (SV-Comp [5]). We remark that the loops in these benchmark programs often contain

non-deterministic choices, which are often used to model I/O environment (e.g., an external function call). As non-determinism is beyond the scope of this work in general, we manually examine each program to check whether our assumption is satisfied or not, i.e., whether the non-determinism is relevant in satisfying the post-condition or not. Only those programs which do not satisfy our assumption are excluded from our experiments. For those which do satisfy our assumption, we replace those non-determinism with random free boolean variables. In total, out of the 323 benchmark programs we gathered, 59 programs are excluded as they do not satisfy their specification; 140 programs are excluded as they do not have non-trivial precondition or postcondition or the loop body contains unsupported constructs like 'break' or 'goto' statement; 59 are excluded as they contain unsupported operations such as array operation; 8 are excluded due to multiple loops; and 15 are excluded due to non-determinism. We also exclude programs which are trivial to prove and copies of the same program. Furthermore, we construct 11 programs (benchmarks[43-53] in the table) due to lack of programs requiring polynomial or disjunctive invariants in these benchmarks. All 53 programs are available at [1].

The parameters in our experiments are set as follows. For random sampling, we generate 10 random values for every input variable of a program from their default ranges. During selective sampling, we generate 2 values for every input variable along the classification boundary. The ratio between random samples and selective samples is thus 5:1, which we consider to be reasonable as selective sampling is slightly more costly. 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 III-D, 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 inequalities, the maximum degree is bounded by 4. In order to give priority to simpler invariants, 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. *Each experiment is executed five times since there is randomness in our approach and we report the median as the result.*

In order to evaluate our approach in learning loop-invariants, we investigate the following research questions:

RQ1 The first research question which we would like to answer is: *does selective sampling help to reduce the number of guess-and-check iterations?* We compare ZILU with and without selective sampling. The results are summarized in Table I. The first column shows the number of the program; the second column shows the type of loop-invariant needed to prove the Hoare triple; and the next six columns show details on verifying the Hoare triple with and without selective sampling. We compare the total number of samples generated and the number of guess-and-check iterations. To reduce randomness, the same set of initial random samples are used in both settings. Each experiment has a time limit of 10 minutes. The winner of each measurement is highlighted boldly.

The results show that ZILU successfully verifies all programs with help of selective sampling, and fails to verify 9 programs without selective sampling. For these 9 programs, ZILU time-outs due to too many guess-and-check iterations. This clearly evidences the usefulness of selective sampling. Furthermore, in 36 cases, selective sampling helps to reduce the number of guess-and-check iterations. Though it rarely happens, due to the randomness in our approach, it may happen that the right invariant is learned by luck with fewer samples. This happens in 3 cases (i.e., 5%) where ZILU without selective sampling has fewer (by 1 or 2) iterations.

We would like to highlight that for 10 programs, ZILU *is able to learn the correct invariant within one guess-and-check iteration with selective sampling*. It is never the case without selective sampling. Furthermore, it happens when the invariant is a linear inequality. 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.

In addition, we observe that ZILU often takes more samples and guess-and-check iterations to learn conjunctive or disjunctive invariants. On average, ZILU takes 1.7, 3, 5.9 and 4.5 guess-and-check iterations to learn linear, polynomial, conjunctive and disjunctive loop-invariants. For conjunctive invariants, more iterations are needed because the algorithm adopted from [48] for learning conjunctive classifiers often requires more samples before convergence. For disjunctive invariants, this is because we need sufficient samples in each partition in order to learn the right invariant.

RQ2 The second research question which we would like to answer is: *does selective sampling incur significant overhead?* This is a valid question as selective sampling requires solving equation systems. In Table I, we show the total execution time of both ZILU with and without selective sampling. It can be observed that the overhead of selective sampling is reasonable. All programs are verified with 2 minutes. A close look reveals that most of the time is spent on classification and selective sampling. For 29 programs, the overall time is reduced with selective sampling, due to a reduced number of guess-and-check iterations. 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

TABLE I: Experiment results

		ZILU w/ Selective Sampling			ZILU w/o Selective Sampling						
benchmark	type	♯sample	♯iter	time(s)	♯sample	♯iter	time(s)	Interproc	CPAChecker	BLAST	InvGen
benchmark01	conjunctive	136	**9**	**8.34**	**63**	13	12.79	✓	3.19	✓	✓
benchmark02	linear	29	**1**	3.06	**12**	2	**2.68**	✓	3.73	✓	✓
benchmark03	linear	62	**2**	**2.88**	**22**	**2**	**2.88**	✓	3.57	✓	✓
benchmark04	conjunctive	197	**5**	8.71	**35**	5	**3.93**	✓	3.26	✓	✓
benchmark05	conjunctive	444	**12**	10.83	**42**	12	**6.66**	✗	3.51	✓	✓
benchmark06	conjunctive	166	**4**	4.48	**34**	4	**3.53**	✓	3.55	✓	✓
benchmark07	linear	**110**	**2**	**3.85**	to	to	to	✗	✗	✗	✗
benchmark08	linear	117	**3**	3.96	**33**	**3**	**3.28**	✗	✗	to	✓
benchmark09	conjunctive	145	**5**	**4.06**	**25**	**5**	4.08	✓	3.20	✓	✓
benchmark10	conjunctive	**213**	**25**	**37.06**	to	to	to	✗	3.26	✓	✓
benchmark11	linear	81	**1**	**5.6**	**22**	2	55.95	✓	3.32	✓	✓
benchmark12	linear	262	**4**	**7.55**	**74**	44	16.22	✓	3.16	✓	✓
benchmark13	conjunctive	240	**6**	15.2	**36**	**6**	**4.54**	✓	3.18	✓	✓
benchmark14	linear	31	**1**	**2.78**	**12**	2	2.95	✓	3.40	✓	✓
benchmark15	conjunctive	**331**	**8**	**77.21**	to	to	to	✓	3.76	✓	✓
benchmark16	conjunctive	87	**3**	3.96	**23**	**3**	**3.63**	✓	3.42	✓	✓
benchmark17	conjunctive	154	**4**	4.2	**34**	**4**	**3.5**	✓	3.63	✓	✓
benchmark18	conjunctive	406	10	65.65	**39**	**9**	**21.31**	✓	3.67	✓	✓
benchmark19	conjunctive	345	**9**	10.34	**40**	10	**8.09**	✓	3.68	✓	✓
benchmark20	conjunctive	148	**4**	**4.79**	**34**	**4**	8.44	✗	✗	✓	✓
benchmark21	disjunctive	**72**	**3**	**91.92**	to	to	to	✗	3.39	✓	✗
benchmark22	conjunctive	158	**6**	**34.46**	**93**	13	44.78	✗	3.71	✗	✗
benchmark23	conjunctive	**170**	**6**	**18.83**	to	to	to	✓	47.82	to	✓
benchmark24	conjunctive	357	9	55.42	**38**	**8**	**26.97**	✗	3.64	✓	✓
benchmark25	linear	31	**1**	**4.3**	**12**	2	4.31	✓	3.05	✓	✓
benchmark26	linear	57	**1**	83.57	**22**	2	**60.2**	✓	3.63	✓	✓
benchmark27	linear	104	**2**	28.34	**33**	3	**3.32**	✓	3.39	✓	✓
benchmark28	linear	66	**2**	5.32	**22**	**2**	**5.07**	✗	✗	✗	✗
benchmark29	linear	36	**1**	**4.69**	**24**	4	6.31	✓	3.09	✓	✓
benchmark30	conjunctive	83	**3**	6.27	**24**	4	**5.76**	✓	3.40	✓	✓
benchmark31	disjunctive	**30**	**2**	**28.06**	40	4	72.18	✗	3.61	✓	✗
benchmark32	linear	33	**1**	**11.6**	**12**	2	15.68	✓	3.41	✓	✓
benchmark33	linear	37	**1**	**9.94**	**12**	2	13.46	✓	3.45	✓	✓
benchmark34	conjunctive	345	9	75.36	**37**	**7**	**27.43**	✓	4.04	✓	✓
benchmark35	linear	39	**1**	**10.45**	**12**	2	13.14	✓	3.61	✓	✓
benchmark36	conjunctive	83	**3**	6.64	**23**	**3**	**6.2**	✓	3.28	✓	✓
benchmark37	conjunctive	104	**4**	**7.84**	**24**	**4**	19.87	✓	3.40	✓	✓
benchmark38	conjunctive	108	**4**	26.06	**24**	**4**	**25.72**	✓	3.51	✓	✓
benchmark39	conjunctive	112	**4**	19.89	**24**	**4**	**18.83**	✗	3.65	✓	✓
benchmark30	polynomial	180	**4**	**34.05**	**25**	5	38.76	✗	✗	✗	✗
benchmark41	conjunctive	263	**5**	17.1	**68**	7	**14.15**	✓	3.70	✓	✓
benchmark42	conjunctive	**334**	**6**	**54.38**	to	to	to	✓	3.33	✓	✓
benchmark43	conjunctive	**214**	**2**	**45.04**	to	to	to	✗	3.42	✓	✓
benchmark44	disjunctive	40	**4**	24.71	**24**	**4**	**16.5**	✓	3.34	✓	✓
benchmark45	disjunctive	**51**	**3**	**5.23**	56	16	7.19	✗	3.31	✓	✗
benchmark46	disjunctive	194	**9**	23.17	**122**	32	**19.91**	✗	3.41	✓	✗
benchmark47	linear	**61**	**1**	**63.55**	to	to	to	✓	3.61	✓	✓
benchmark48	linear	297	**3**	25.93	**34**	4	**17.03**	✓	3.77	✓	✓
benchmark49	linear	80	**2**	37.52	**34**	4	**16.77**	✓	3.36	✓	✓
benchmark50	linear	90	**2**	**13.63**	**22**	**2**	14.83	✓	3.51	✓	✓
benchmark51	polynomial	48	**2**	**12.2**	**42**	4	22	✗	3.25	✓	✓
benchmark52	polynomial	**180**	**4**	**69.99**	to	to	to	✗	✗	✗	✗
benchmark53	polynomial	176	**3**	**61.71**	**105**	5	63	✗	✗	✗	✗

iteration. Comparing the number of samples generated for each program, it can be observed that ZILU with selective sampling often generates more samples. This is expected as we generate multiple samples for each call of $selectiveSampling$, whereas only one sample is generated during the verification phase. This is because sampling through verification has, in general, a high cost and we aim to avoid it as much as possible.

RQ3 The third research question is: does ZILU outperform existing state-of-the-art program verification tools on verifying these programs? Ideally, we would like to compare with those tools reported in [48], [47], [46], [44], [19], [18]. Unfortunately, those tools are not maintained. We instead compare ZILU with four state-of-the-art tools on loop-invariant generation and program verification. In particular, Interproc [30] is a program verifier which 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. CPAChecker [7] is a state-of-the-art program verifier. The CPAChecker which we use in this work is the version used for SV-COMP 2017 [5]. Note that CPAChecker supports a variety of verification methods and it

is configured in the exact same way as in SV-COMP 2017[3]. BLAST is a software model checker based on counterexample-guided abstraction refinement [6]. Lastly, InvGen [25] is a tool which aims to generate linear arithmetic invariants, using a combination of static and dynamic analysis techniques.

The results are shown in the last 4 columns of Table I where ✓ means that the Hoare triple is verified and ✗ means either it outputs no conclusive results or false positives. We remark that because the tools use approaches which are different from each other, the comparison should be taken with a grain of salt. Interproc and InvGen are very efficient in handling the programs, i.e., within 1 second for each program, and thus we skip the verification time. BLAST is similarly efficient except that it timeouts in two cases. We show the timed taken by CPAChecker in case it successfully verifies the program.

We have the following observation based on the experiment results. First, for all 53 programs, ZILU is able to find a loop-invariant which proves the Hoare triple. In comparison, Interproc failed in 18 cases; CPAChecker failed in 7 cases; BLAST failed in 8 cases; and InvGen failed in 10 cases. We note that this comparison might be not fair, as we have not compared other tools on the excluded programs. More precisely, for the programs that are compatible with input restrictions of ZILU, ZILU works best compared with other tools. Secondly, existing tools often complement each other. For instance, BLAST successfully proves all programs which require disjunctive loop-invariant, whereas it failed in several cases where a polynomial loop-invariant is required. In contrast, programs which require disjunctive loop-invariants are often challenging for other tools (except ZILU). Moreover, due to the dynamic nature, ZILU often requires more time. Nonetheless, we consider that ZILU is relatively efficient. For all 53 programs, ZILU finishes the proof within 92 seconds.

V. RELATED WORK

The closest related work are those guess-and-check approaches on invariant generation. In [48], the authors proposed to generate samples through constraint solving and learn loop-invariants based on *SVM* classification. In comparison, ZILU learns more expressive invariants in the form of polynomial inequalities or their disjunctions and conjunctions. More importantly, we apply active learning with selective sampling so as to overcome the limitation of too few samples or too many guess-and-check iterations. In [47], the authors proposed to apply PAC learning techniques for invariant generation. It has been demonstrated that their approach may learn invariants in the form of arbitrary boolean combinations of a given set of propositions (under certain assumptions). In [46], the authors developed a guess-and-check algorithm to generate invariants in the form of the algebraic equation. It learns invariants of polynomial form by operating the null space operation on matrix. In [44], the authors proposed a framework for generating invariants based on randomized search. In particular, their approach has two phases. In the search phase, it uses randomized search to discover candidate invariants and uses a checker to either prove or refute the candidate in the validate phase. In [18], Pranav Garg *et. al* proposed to synthesize invariants by learning from implications along with positive and negative samples. They further extend their approach by modifying existing decision tree classification algorithm with heuristics adopted from [20]. In this way, they could cope with implication better and, as a result, handle invariants of combination of conjunctions and disjunctions in theory. One limitation of their work is that the terms in the decision tree (e.g., the propositions) must be pre-defined.

Compared with the above-mentioned work, ZILU improves loop-invariant generation through active learning with selective sampling, so as to avoid applying the invariant checker as much as possible. To the best of our knowledge, ZILU is the first to combine selective sampling with invariant inference. In particular, in the guessing phase, we additionally adopt a learn-and-refine iteration which improves the invariant candidates through classification and selective sampling. In comparison, other guess-and-check approaches solely rely on the checkers to improve invariant candidates. Furthermore, we show ZILU can be extended easily to learn disjunctive loop-invariants through data partitioning and classification.

Lastly, our approach can be extended to learn other forms of classifiers classification algorithms. For instance, in principle, any arbitrary mathematical classifiers can be learnt using methods like *SVM* with kernel methods [29], but interpreting the output models is challenging. Nonetheless, we focus on invariants in the form of polynomial inequalities or conjunctions/disjunctions of polynomial inequalities in our evaluation. The experiment results show that our approach effectively learns loop-invariant for proving a set of benchmark programs and complements the existing approaches.

Besides the guess-and-check approaches, some alternative approaches have been proposed for invariant generation. Examples include those based on abstraction interpretation [13], [36], [31], those based on counterexample-guided abstraction refinement [28], [3], [11] or interpolation [27], [34], [35], [33], and those based on constraint solving and logical inference [25], [12], [24], [17]. These approaches depend on constraint solving and thus suffer from scalability. For instance, the work in [36], [31], [25] is restricted to generate invariants in abstract domains for which constraint solving is manageable.

VI. CONCLUSION

In this work, we propose a systematic approach to learn loop-invariants based a combination of selective sampling and guess-and-check. As for future work, we would explore methods for learning more expressive loop-invariants as well as methods for discovering new features for our classification.

VII. ACKNOWLEDGEMENTS

We greatly appreciate the anonymous reviewers for their insightful feedback on the early draft of this paper. This research is supported by MOE research grant "T2MOE1704".

[3]We thank the help from a researcher in the SV-COMP evaluation team.

REFERENCES

[1] ZILU repo. https://github.com/lijiaying/zilu, 2017.

[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. Software verification with validation of results. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–349. Springer, 2017.

[6] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[7] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *CAV*, pages 184–190, 2011.

[8] 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.

[9] 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.

[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] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 69–87, 2014.

[19] P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 499–512, 2016.

[20] P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In *ACM SIGPLAN Notices*, volume 51, pages 499–512. ACM, 2016.

[21] R. Gilad-Bachrach, A. Navot, and N. Tishby. Query by committee made real. In *NIPS*, pages 443–450, 2005.

[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 *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 634–640, 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] V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *VMCAI*, pages 229–244, 2009.

[32] D. D. Lewis and W. A. Gale. A sequential algorithm for training text classifiers. In *SIGIR Forum*, pages 3–12, 1994.

[33] S.-W. Lin, J. Sun, T. K. Nguyen, Y. Liu, and J. S. Dong. Interpolation guided compositional verification (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 65–74. IEEE, 2015.

[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] B. Settles. *Active Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

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

[45] R. Sharma, I. Dillig, T. Dillig, and A. Aiken. Simplifying loop invariant generation using splitter predicates. In *CAV*, pages 703–719, 2011.

[46] 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.

[47] 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.

[48] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *Computer Aided Verification*, pages 71–87. Springer, 2012.

[49] 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.

[50] S. Tong and D. Koller. Support vector machine active learning with applications to text classification. *Journal of Machine Learning Research*, 2:45–66, 2001.