Synthesize Inductive Invariants by K-means++ and Support Vector Machine

Shengbing Ren
School of software, Central South University
Changsha, China
e-mail: rsb@csu.edu.cn

Xiang Zhang
School of software, Central South University
Changsha, China
e-mail: 455290289@qq.com

Abstract—**The problem of synthesizing adequate inductive invariants lies at the heart of automated software verification. With the rapid development of artificial intelligence, the state-of-the-art machine learning algorithms for invariants synthesis have gradually been developed and have shown its excellent performance. However, when positive samples are distributed among multiple clusters, the algorithm for intersection of half space based on SVM can not generate the candidate invariants. In order to solve this problem, a new modified algorithm, based on and k-means++ and SVM, is proposed. After sampling, our algorithm adopts k-means++ to cluster the positive samples. Then this algorithm uses SVM to distinguish each positive sample cluster from all negative samples to compute the candidate invariants. Finally, a set of theories founded on Hoare logic are adopted to check whether the candidate invariants are true invariants. If the check succeeds, the true invariants have been found. Otherwise, we should collect more samples and repeat our algorithm. Experimental results show that our algorithm can compute the candidate invariants when positive samples are distributed among multiple clusters.**

Keywords-software verification; invariants synthesis; intersection of half space; k-means++; SVM; Hoare logic

I. INTRODUCTION

With the increasing size and complexity of software, it becomes more and more difficult and complex to verify the correctness of the software. Thus, how to ensure the correctness of software has aroused enough attention [1, 2]. In order to handle this problem, one of the popular technique is software verification [3].

However, the limitations of manual verification are becoming more and more obvious, and software verification technology needs innovative development. In recent years, the automated technology and tools for software verification have gradually become an important research direction [4]. The purpose of this paper is to propose a method applied to the field of automated program verification.

In the process of automated program verification, synthesizing inductive invariants plays a key role [5]. An invariant means that it is closed with respect to the transition relation of the program, and it guarantees that any execution of a statement in the program changes from a state that belongs to the invariant region to other state that also belongs to the invariant. Once adequate inductive invariants have been found, the problem of software verification can be reduced to logical validity of verification conditions, which are solved with the advances in automated logic solvers [6].

In the past, people put forward a lot of solutions to synthesize inductive invariants including model checking, abstract interpretation [7], Craig's interpolation [8]. Although, these techniques are able to compute invariants, they have their own inherent hardness, accompanied by certain limitations. For example, model checking can successfully synthesize invariants when the program has a finite state-space or the paths in the program are bounded. However, for programs over an infinite domain, for e.g. integers, with unbounded number of paths in the program, model checking is doomed to fail [6]. With the rapid development of artificial intelligence, the state-of-the-art machine learning algorithms, such as support vector machine (SVM) [9], decision trees [10, 11] and winnow algorithm [12], applied to invariants synthesis have gradually been explored in recently years. Machine learning algorithms have earned enough attention, and more and more people are devoted to the research of algorithms and tools. Our method also adopts machine learning algorithm to synthesize invariants.

In this paper, we develop a new technique based on k-means++ [13], SVM [14] and Hoare logic [15, 16] to synthesize invariants. Specially, our method have ability to synthesize invariants although the positive samples are distributed among multiple clusters. At first, we collect samples and then cluster positive samples by k-means++. The second step of our method is to separate positive samples from negative samples and get the hyperplane equations output by the algorithms for intersection of half space (IOH) [14]. The final step is to check the correctness of our result by Hoare logic.

The rest of the paper is organized as follows. We simply introduce our technique by the way of two examples in Sect. 2. In Sect. 3, we describe some necessary materials including sampling, k-means++, algorithms for intersection of half space and Hoare logic. And then we give a detailed process of our algorithm in Sect. 4. In Sect. 5, we show the results and verify the effectiveness of our algorithm. Finally, Sect. 6 concludes with some directions for future work.

II. MOTIVATING EXAMPLES

A. Example I

Considering the program in Figure 1, it has two integer variables x and y. After passing two loops, the value of x and y have changed. In the end, we should check whether the error() state can be reached by the value of y. If the error() state can't be reached, the initial values of x, y and their

978-1-7281-2871-9/19/$31.00 ©2019 IEEE

metabolic values in the program paths are good states. If we assign arbitrary values to x and y, then the error() state is reached after executing the program. Only the initial values of x and y, except their values in the path, are bad states. Suppose we consider a path that goes through the two loops once. We sample these two points {(0, 0), (1, 1)} as good states and points {(0, 1), (1, 0)} as bad states. Figure 1 gives the codes and plots the distribution of these four points. The solid points represent positive samples and the hollow points represent negative samples.

By observing the program, we find the tendency of the variables is linear, thus only linear inequalities, rather than nonlinear inequalities, can better characterize the intrinsic properties of this program. So we adopt the machine learning algorithm of linear kernel SVM to obtain the linear inequalities. Unfortunately, the distribution of this four points can not be separated by using linear kernel SVM once. However, there exist two linear inequalities $2y < 2x + 1$ and $2y > 2x - 1$ which are able to represent invariants (see the two straight lines in Figure 1). If we choose the samples {(0, 1), (0, 0), (1, 1)} and use linear kernel SVM, we can get the invarints $2y < 2x + 1$. Similarly, if we choose the samples {(1, 0), (0, 0), (1, 1)}, we can get the invarints $2y > 2x - 1$. And then we take the intersection of two inequalities. The result, $2y > 2x - 1 \wedge 2y < 2x + 1$, is out. Because the type of x and y is integer, our result can be equivalent to $y \geq x \wedge y \leq x$. And then we can regard x = y as final result. Finally, by a set of theories of Hoare logic, it can be checked that x = y is a true invariant, which is sufficient to prove the error() state is unreachable.

```
foo(){
   int x = 0;
   int y = 0;
   while (*)
    { x ++; y ++; }
   while(x!=0)
   {x --; y --;}
   if (y != 0)
   { error(); }
}
```

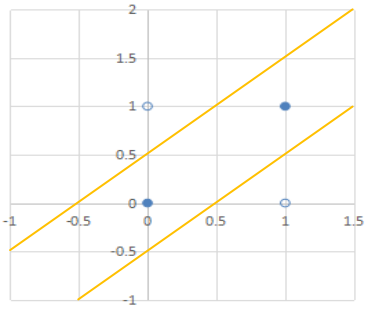

Figure 1. Motivating Example I.

B. Example II

Considering the program in Figure 2, we sample in the same way as example I. For instance, if we assume the values of x and y are (0, -3), then the states we reach are {(-1, -2), (-2, -1), (-3, 0)} and thus these are all good states. Similarly, if we define their values as (-2, 2), then the states we reach are {(-1, 1), (0, 0)} which violate the assertion and thus (-2, 2) is a bad state. And if we directly define their values are (0, 0), then the loop will not run and the value of x is 0 which violates the assertion as well. We plot samples in a coordinate system. Then we arbitrarily choose one negative sample and all positive samples and continue to use linear kernel SVM. Of course, we can't get a correct result, because positive samples are distributed among multiple clusters and positive and negative samples can't be separated. The solution is that we use k-means++ and cluster positive samples first. The value of k in k-means++ is two, which is the most suitable value with this example. Then we separate every positive samples cluster respectively from all negative samples by using SVM. Then the result is $y > -x + 0.5 \vee y < -x - 0.5$. It is worth noting that all negative samples, rather than one negative sample, are chosen. This is different from example I. For separating every positive samples cluster from all negative samples, using SVM once is workable under this kind of circumstances. Similarly, we should check whether our result is correct in the end.

```
fun(){
   int x, y;
   assume x = 0 ∧
          y != 0;
   while (y != 0) {
     if(y < 0){
       x = x - 1;
       y = y + 1;
     }else{
       x = x +1;
       y = y - 1;
     }
   }
   assert  x != 0;
}
```

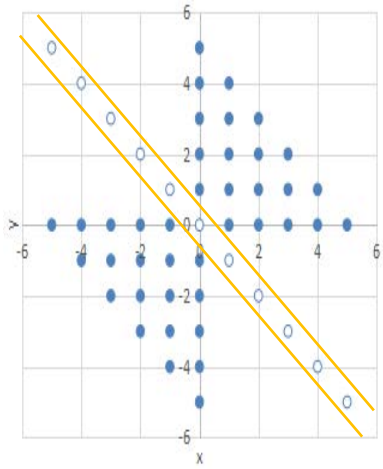

Figure 2. Motivating Example II.

III. PRELIMINARIES

A. Sampling

If we regard a program as a transition system, the states of program can be split two states - good states and bad states - when we execute the program. The good states are states that include the initial states of the program and can satisfy a specified safety specification while bad states can not satisfy. If the program is correct, then there is no transition sequence from an initial state to a bad state. *Reach* is the set of reachable states and *I* is an adequate inductive invariant used to distinguish these two states [6] (see Figure 3). If we can assure any state in the program belongs to *Init* and *Reach*, this program is correct.

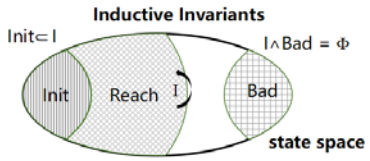

Figure 3. Program state space.

The above interpretations guide us how to sample. A good state is defined as any state that the program could conceivably reach when it is started from a state consistent with the precondition [11]. In other words, the initial value and their changed values which can make bad states

unreachable are both positive samples. Similar, a bad state is defined as states, except the states in program path, that will make assertion fail after executing the program. Thus, the values that will make assertion fail represent the negative samples.

B. K-means++

K-means++, proposed by David Arthur and Sergei Vassilvitskii in paper [13], is a kind of clustering algorithm on the basis of k-means. When we adopt k-means++ algorithm, the most difficult thing is how to choose the most suitable value of k which is related to the number of clusters. In our algorithm, we gradually increase the value of k by 1 and find the value of the optimal k by the way of iterating. The specific method will be discussed in Sect. 4. It is because of this way, IOH is compatible with our algorithm, and the most suitable value of k will also be found at the first time.

C. Algorithm for Intersection of Half Space

In the limitation where only linear kernel SVM can be used with the linear tendency of variables, we adopt the algorithms for intersection of half space when the samples can't be separated by using linear kernel SVM only once. We give out the pseudo code of the IOH and explain the steps in detail (See Algorithm 1).

Algorithm 1 IOH

```
Input:   Positive sample: X+
         Negative samples: X-
         Cost parameter:c := 1000
Output:
         Candidate Invariants::H
1:         H := true
2:         Misclassified := X−
3:         While | Misclassified | ≠ 0 :
4:                 Arbitrarily choose b from Misclassified
5:                 h := Process(SVM(X+, {b}), X+, X−)
6:                 ∀ b' ∈ Misclassified s.t. h(b') < 0
7:                 Remove b' from Misclassified
8:                 H := H ∧ h
9:         End While
10:        Return H
```

First, we have three samples sets *X+*, *X-, Misclassified* and a hyperplane set *H*. *X+* and *X-* are the sets of positive samples and negative samples respectively. *Misclassified* represents the samples misclassified by the hyperplane *H*. At the beginning, the set of *H* is empty and all the samples have not been classified. Thus, all the positive samples have been classified correctly and all the negative samples have been classified by mistake. The set of *Misclassified* is equal to *X-*. Secondly, while the set of *Misclassified* is not empty, we choose a sample *b* from *Misclassified* randomly and use SVM to get a hyperplane *h* with samples *b* and *X-*. Then we remove the samples classified correctly by *h* from *Misclassified*, and then define *H* as $H \wedge h$. We repeat the above steps until all samples have been classified correctly. In others word, the end condition of the loop is that *Misclassified* is empty. Finally, *H* is the result we try to find.

D. Hoare Logic

A set of systematic ideas of machine learning are constructing a model by training set, adjusting parameter by validation set and checking whether the model is reliable by test set. The process of sampling and using machine learning algorithms to compute invariants can be regarded as process of modeling. When we use linear kernel SVM, all the parameters that we should adjust are only cost parameter c. In our experiment, we give *c* a constant value to avoid misclassification. Therefore, the validation set is not needed. Not like machine learning, our method adopts a set of theories based on Hoare logic to check candidate invariants.

Hoare logic, also known as Floyd-Hoare logic, is a formal system developed by British computer scientist C. A. R. Hoare. The purpose of this system is to provide a set of logical rules for the correctness of computer programs using strict mathematical logic reasoning. The central feature of the Hoare logic is the Hoare triple :

$$\{P\}C\{Q\} \tag{1}$$

In Hoare triple, P represents pre-condition, Q is the post-condition and C means the code segment. If a program is correct, it means that when P holds, after the execution of C, Q holds.

As the program with loop, the invariant is loop invariant. Loop invariant synthesis is a huge challenge [15, 16]. It is an assertion that has been held since the beginning of the execution of the loop until the end of the loop. Let's give a formal description of the program with loops:

$$\{P\}\text{while}B\{C\}\{Q\} \tag{2}$$

B represents loop condition and C is the code in loop. On the basis of existing theories, we can confirm that I is a correct invariant as long as $\{P\} \Rightarrow \{I\}$, $\{I \wedge B\}C\{I\}$ and $\{I \wedge !B\} \Rightarrow \{Q\}$. The sign of $\Rightarrow$ means implicating rather than inferring. For example, x = y can not infer x >= y, but x = y implicates x >= y. In other words, while the condition x = y holds, x >= y also holds.

In our experiments, the I in program of Figure 1 is x = y, and P is x = y = 0. $\{P\} \Rightarrow \{I\}$ without any doubt. $\{I \wedge B\}C\{I\}$ holds, because the changes of x and y are the same in C. B is x != 0, so !B is x = 0. $I \wedge !B$ means x = y and x = 0. We can get a conclusion that y = 0 which can avoid program reaching to the statement of error(). Thus, x = y is the true invariant of this program.

If we found that I is an error invariant by theorem prover, we need to sample more. Specifically, if we just sample three points{(0, 1), (0, 0), (1, 1)}, we can get an invariant y < x - 0.5 by using our algorithms. Because the type of variables x and y is integer, the invariant is equal to x >= y. $\{P\} \Rightarrow \{I\}$ holds, but $\{I \wedge !B\} \Rightarrow \{Q\}$ does not hold. It means some bad states are contained by I, so we should collect more samples and repeat our algorithms.

IV. K-MEANS++ WITH IOH

IOH is workable, but there are still problems waiting to be solved. Considering the example II, after sampling, it is found that the positive samples are distributed among multiple clusters. IOH just choose one negative sample, it is

surprised to discover IOH is infeasible under this situation. Thus, our method based on k-means++ and IOH (K-IOH) emerges as the times required for solving this kind of problem (see Algorithm 2) . At first, we follow a specific program to sample positive samples and negative samples. Then, K-IOH defines the value of k as 1 at the beginning, and then adopts k-means++ algorithm to cluster the positive samples as k clusters. Next, our algorithm uses linear kernel SVM to separated samples. If the samples can be separated, we get the candidate invariants. Otherwise, IOH is used to distinguish each positive samples cluster from all negative samples to get the intersection of hyperplane equations. If IOH is valid, the candidate invariants are found. If not, this algorithm lets the value of k add 1 and clusters the positive samples again. Then repeat algorithm till it can distinguish and get the candidate invariants. Finally, we should check whether the candidate invariants are true invariants. If the check succeeds, we get the final result. Otherwise, we should collect more samples and repeat our algorithm.

The condition for changing the value of k is that IOH can not get the candidate invariants, but this description is not an algorithmic language. In fact, in the process of algorithm implementation, we need to change the value of k as long as the capacity of *Misclassified* is not reduce. Every time, we choose one negative sample and then use SVM to distinguish the one negative sample and positive samples (or positive samples cluster). That is to say, every time we can at least separate out one negative sample. If the capacity of *Misclassified* reduces, we continue to execute the algorithm. Otherwise, we stop IOH and let the value of k add 1.

Algorithm 2 K-IOH

```
Input:    Positive sample: X+
          Negative samples:  X-
          Cost parameter:c :=  1000
Output:
          True Invariants: H
1:  k := 1
2:  Cluster (x+, k) , obtain X: {X1+,  X2+,  .... , Xk+}
3:  If (SVM can be used) :
4:      For i=1 to k ( i = k is allowed):
5:          h := Process ( SVM( Xi+, {b} ) )
6:          H := H ∨ h
7:      End For
8:  Else:
9:      For i=1 to k ( i = k is allowed) :
10:         Misclassified := X-
11:         While | Misclassified | ≠ 0 :
12:             Arbitrarily choose b from Misclassified
13:             h := Process( SVM ( Xi+, {b} ) )
14:             ∀b' ∈ Misclassified s.t. h(b') < 0
15:             Remove b' from Misclassified
16:             If (| Misclassified | == | Misclassified |'):
17:                 k := k + 1
18:                 Goto 2
19:             Else:
20:                 Hi := h ∧ hi
21:         End While
22:         H := H ∨ Hi
23:     End For
24: If ( H is indeed invariants) :
25:     Return H
26: Else:
27:     Add more samples to X+ and X-
28:     Goto 2
```

V. EXPERIMENTAL EVALUATION

We have implemented a prototype version of the algorithm described in this paper in the compute language of Python. First, we obtain constraint expression file whose type is txt by using a Bounded Model Checker (CBMC, we can download it in http://www.cprover.org/cbmc). CBMC is applicable to C and C++ programs and, just right, all of our experimental programs are C programs. Secondly, according to the constraint expression file, we generate samples by using the Z3 SMT solver for constraint solving. Finally, to realize our algorithm and show results, numpy, sklearn, codecs, re and matplotlib packages of Python are adopted.

The k-means++ and SVM in K-IOH are supported by off-the-shelf sklearn packages of Python. There is an important problem concerning the value of the cost parameter c of SVM algorithm should take. In order to guarantee that under certain conditions the programs will never violate assertion, our classifier is not allowed to misclassify. A low value of c allows the generated classifier to make errors on the training data. So, we assign a very high value to c (1000 in our experiments). We conduct all of the following experiments on a core I5 CPU with 8GB of RAM running Windows 10.

The results are shown in table 1 and table 2. *File* is the name of the program, and we can find it in the reference behind the name; *LOC* are code lines; *Invariants* are the invariants of the program output by using IOH. Thereinto, *Fail* means that finding invariants by IOH is failed; *OurInvariants* are the invariants of the program output by using K-IOH; *Num* means the total number of the samples including positive samples and negative examples; *Times* is the running time of the IOH; *OurTimes* is the running time of the K-IOH. The statistical time is from input samples to output candidate invariants; *K* is the value of k in K-IOH; *It* means the iterations of the loop in every program, and * means that our experiment do not limit the number of loops which is determined by specific program.

TABLE I. EXPERIMENT RESULTS

File	LOC	Invariants	OurInvariants	It
1.fig1a [14]	14	(y > x - 0.5) ∧ (y < x + 0.5)	(y > x - 0.5) ∧ (y < x + 0.5)	1
2.fig1a [14]	14	(y > x - 0.5) ∧ (y < x + 0.5)	(y > x - 0.5) ∧ (y < x + 0.5)	4
3.ex1 [17]	22	xa + 2ya > 2	xa + 2ya > 2	1
4.fig2 [17]	17	(y < 2x + 0.5) ∧ (y > -x - 1)	(y < 2x + 0.5) ∧ (y > -x - 1)	1
5.fig1 [18]	10	Fail	(y > -x + 0.5) ∨ (y < -x - 0.5)	*
6.fig4 [18]	8	Fail	(i < 0.5) ∨ (j < i + 0.5)	*
7.fig1 [19]	8	Fail	(y > 0.5) ∨ (x <-0.5)	*
8.fse6 [4]	8	(y > -0.5) ∧ (x > -0.5)	(y > -0.5) ∧ (x >-0.5)	*

By observing the results of the experiment, we can get the following conclusions:

- When the positive samples are distributed only in one cluster, K-IOH and IOH both can find the invariants with the unanimous results.
- However, only K-IOH can find the invariants when the positive samples are distributed among multiple clusters.
- When the positive samples are distributed only in one cluster, the running time of K-IOH is closed to IOH. Generally speaking, the running time of K-IOH is slightly higher than IOH, but sometimes slightly lower. It is reasonable that K-IOH is slightly higher, because K-IOH is more complex and should use k-means++ algorithm extra. The situation K-IOH is slightly lower is unexpected, but it is still reasonable, because when they use SVM, they choose one positive sample randomly. This randomness results that the iterations of algorithms can't be controlled, although it does not have a great impact on the results. Removing the randomness is the point we will research in the future.

TABLE II. RUNNING TIME COMPARISON

File	Num	Times(ms)	OurTimes (ms)	K	It
1.fig1a	4	1.578	1.627	1	1
2.fig1a	20	3.311	3.912	1	4
3. ex1	20	2.529	2.488	1	1
4. fig2	20	3.499	4.559	1	1
5. fig1	50	Fail	43.865	2	*
6. fig4	20	Fail	3.128	1	*
7. fig1	56	Fail	159.152	2	*
8.fes6	16	1.989	2.594	1	*

VI. CONCLUSION AND FUTURE WORK

We have shown that synthesizing invariants by machine learning algorithm can be profitable. In particular, when the positive samples are distributed among multiple clusters, the K-IOH based on IOH and k-means++ is valid. It is worth noting that when we consider synthesizing invariants as a problem of classification based on machine learning algorithms, we must ensure the correctness of result output by model. In this paper, we use the theory method based on Hoare logic to prove the correctness of the invariants. Through our experiments, we have also verified that K-IOH is not only compatible with IOH, but can solve the problem while IOH can't solve.

As future work, we would like to modify our algorithm to eliminate the randomness and to make it more stable and robust. As the tendency of the program variables is non linear, we believe SVM also can compute the invariants with the form of nonlinear. Computing nonlinear invariants is the research we would like to be dedicated to in the future.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their precious comments. We thank Xiangdang Liao, Ruliang Xie, Liujie Hua, and Lingling Ye for their helpful discussion.

REFERENCES

[1] M. Stavnycha, H. Yin, "A Large-scale Survey on the Effects of Selected Development Practices on Software Correctness," International Conference on Software and System Process. ACM, 2015, pp. 117-121.

[2] J. T. Mcdonald, T. H. Trigg, Roberts C E, et al. "Security in Agile Development: Pedagogic Lessons from an Undergraduate Software Engineering Case Study," Cyber Security Symposium, Springer International Publishing, 2015, pp. 127-141.

[3] V. D'Silva, D. Kroening, G. Weissenbacher. "A Survey of Automated Techniques for Formal Software Verification," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2008, vol. 27, pp. 1165-1178.

[4] B. S. Gulavani, T. A. Henzinger, "SYNERGY: A New Algorithm for Property Checking," ACM Sigsoft International Symposium on Foundations of Software Engineering, 2006, pp. 117-127.

[5] Y. Vizel, A. Gurfinkel, S. Shoham, "IC3 - Flipping the E in ICE," International Conference on Verification, Model Checking, and Abstract Interpretation. Springer International Publishing, 2017, pp. 521-538.

[6] P. Garg, "Learning-based Inductive Invariant Synthesis," Univerisity of Illinois Urbana-Champaign, 2015.

[7] P. Cousot, R. Cousot R, L. Mauborgne, "Theories Solvers and Static Analysis by Abstract Interpretation," Journal of the ACM, 2012(6):31-1-31-56.

[8] K. L. Mcmillan, "Interpolation and SAT-Based Model Checking," Computer Aided Verification. Springer Berlin Heidelberg, 2003, pp. 1-13.

[9] C. Cortes, V. Vapnik, "Support-vector networks," Machine Learning, 1995, 20(3): 273-297.

[10] P. Garg, D. Neider, P. Madhusudan, et al, "Learning Invariants Using Decision Trees and Implication Counterexamples," Acm Sigplan Notices, 2015, vol. 51, pp. 499-512.

[11] S. Krishna, C. Puhrsch, T. Wies, "Learning Invariants Using Decision Trees," Computer Science, 2015, pp. 44-59.

[12] Nick Littlestone, "Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm," Machine Learning, 1987, 2(4), pp. 285‑318.

[13] D. Arthur, S. Vassilvitskii, "K-means++: the Advantages of Careful Seeding," Eighteenth Acm-Siam Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, 2007, pp. 1027-1035.

[14] Rahul Sharma, V. Aditya, Nori, Alex Aiken, "Interpolants as Classifiers," Lecture Notes in Computer Science, 2012, vol. 7358, pp. 71-87.

[15] R. Sharma, A. Aiken, "From Invariant Checking to Invariant Inference Using Randomized Search," International Conference on Computer Aided Verification. Springer-Verlag New York, Inc. 2014, pp. 88-105.

[16] R. Sharma, S. Gupta, B. Hariharan, et al, "Verification as Learning Geometric Concepts," Static Analysis. Springer Berlin Heidelberg, 2013, pp. 388-411.

[17] B. S. Gulavani, "Automatically Refining Abstract Interpretations," Theory and Practice of Software, International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems, 2008, pp. 443-458.

[18] J. Li, J. Sun, L. Li, et al, "Automatic Loop-invariant Generation and Refinement Through Selective Sampling," International Conference on Automated Software Engineering. IEEE, 2017, pp. 782-792.

[19] S. Gulwani, S. Srivastava, "Program Analysis as Constraint Solving," ACM Sigplan Conference on Programming Language Design and Implementation, 2008, pp. 281-292.