Synthesizing Conjunctive and Disjunctive Linear Invariants by K-means++ and SVM

Shengbing Ren and Xiang Zhang
School of Computer Science and Engineering, Central South University, China

Abstract: *The problem of synthesizing adequate inductive invariants lies at the heart of automated software verification. The state-of-the-art machine learning algorithms for synthesizing invariants have gradually shown its excellent performance. However, synthesizing disjunctive invariants is a difficult task. In this paper, we propose a method k++ Support Vector Machine (SVM) integrating k-means++ and SVM to synthesize conjunctive and disjunctive invariants. At first, given a program, we start with executing the program to collect program states. Next, k++SVM adopts k-means++ to cluster the positive samples and then applies SVM to distinguish each positive sample cluster from all negative samples to synthesize 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 candidate invariants fail the check, we should sample more states and repeat our algorithm. The experimental results show that k++SVM is compatible with the algorithms for Intersection Of Half-space (IOH) and more efficient than the tool of Interproc. Furthermore, it is shown that our method can synthesize conjunctive and disjunctive invariants automatically.*

Keywords: *Software verification, conjunctive invariant, disjunctive invariant, k-means++, SVM.*

Received September 5, 2018; accepted January 28, 2020
https://doi.org/10.34028/iajit/17/6/3

1. Introduction

With the increasing size and complexity of software, it is more difficult and complex to verify the correctness of the software. Thus, how to ensure the correctness of software has aroused enough attentions [19, 26, 27]. In order to handle this problem, one of the popular techniques is the software verification [4].

However, the limitations of manual verification are becoming more and more obvious, and software verification technology needs innovative development. In recent years, the automated techniques and tools for software verification have gradually become an important research direction [10].

In the process of automated program verification, synthesizing inductive invariants plays a key role [28]. 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. 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 [7, 8]. What's more, invariants also can be useful for compiler optimization, program understanding [18], bug detection [5].

In the past, people have put forward a lot of solutions to synthesize inductive invariants including model checking, abstract interpretation [3], Craig's interpolation [20]. 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, such as integers, with unbounded number of paths in the program, model checking is doomed to fail [7]. With the rapid development of artificial intelligence, the state-of-the-art machine learning algorithms, such as Support Vector Machine (SVM) [17, 24], decision trees [8, 15], and learning using Examples, Counter-examples, and Implications (ICE) [6], have been applied to synthesize invariants in recently years.

Recently, a great deal of approaches are based on guess-and-check model to synthesize invariants [6, 8 17, 22, 23, 24, 25]. Roughly speaking, those models regard the problem of synthesizing invariants as two parts, learner and teacher. The learner synthesizes candidate invariants and the teacher checks whether the candidate invariants are true invariants by a set of theories founded on Hoare logic in each round. If not, this model should give more details for learner to revise candidate invariants until the candidate invariants pass the checking. A common problem with the guess-and-check model is that their effectiveness is often limited by the samples collected in the first phase [17]. The paper [17] gives us an ideal that we can use selective samples to improve the efficiency.

In this paper, we develop a new method, called k++SVM, based on k-means++ [1], SVM [2], Hoare logic [13, 23, 25], selective samples and the guess-and-check model to synthesize invariants. At first, we collect samples and then cluster positive samples by k-means++. Secondly, SVM is adopted to separate positive samples from negative samples and get the hyper plane equations. The final step is to check the correctness of our results by Hoare logic.

1.1. Our Contributions

This paper makes the following contributions:

- We propose a method based on k-means++, SVM, Hoare logic, selective samples and the guess-and-check model to synthesize not only conjunctive invariants but disjunctive invariants as well.
- We show that our method has the ability to automatically synthesize conjunctive and disjunctive invariants. Actually, automatically synthesizing conjunctive and disjunctive invariants is a big challenge. Our method solves this problem in a unique way.
- We have implemented our method in Python for synthesizing invariants and we compare it with the algorithm Intersection of Half-space (IOH) and a tool named Interproc. The experimental results show that our method is compatible with IOH. Furthermore, it is shown that our method can synthesize conjunctive and disjunctive invariants automatically while the IOH and Interproc cannot.

1.2. Organization

The rest of the paper is organized as follows. We simply introduce our method by the way of two examples in section 2. Next, section 3 reviews related work. In section 4, we describe necessary material including sampling, k-means++, algorithms for intersection of half-space and Hoare logic. And then we give a detailed process of k++SVM in section 5. In section 6, we show the experimental results and verify the effectiveness of our algorithm. Finally, section 7 concludes with some directions for future work.

2. The Motivating Examples

2.1. Conjunctive Invariants

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 bad state error() can be reached by the value of y. If the error() state cannot be reached, the initial values of x, y and their metabolic values in the program paths are good states. If we give a arbitrary values of x, y, then the error() state is reached after executing the program. Only the initial values of x, 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 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 cannot be separated by using linear kernel SVM once. However, there exist two linear inequality 2y<2x+1 and 2y>2x-1 which are able to represent invariants (see the two straight lines in Figure 1). Our solution is that, firstly, we use k-means++ to cluster the positive samples (0,0), (1,1) as one cluster in the condition the value of k is 1. If we choose the negative sample (0, 1) with positive sample cluster and use linear kernel SVM, we can get the invariant 2y<2x+1. Similarly, if we choose another negative samples (1, 0) with positive sample cluster and use SVM, we can get the invariants 2y>2x-1. After we take the intersection of two inequalities, we can get the result 2y>2x-1∧ 2y<2*x*+1. 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 an invariant which is sufficient to prove the error() state is unreachable.

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

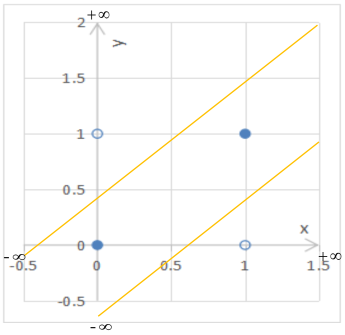


Figure 1. Conjunctive invariants.

2.2. Disjunctive Invariants

Considering the program in Figure 2, we sample in the same way as example 1. 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.

```
fun(){
sint 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);
}
```

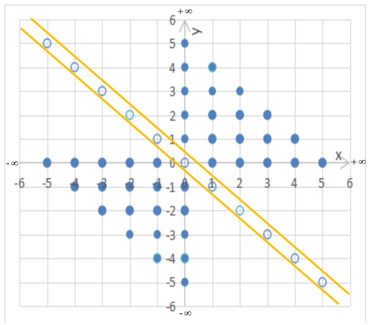


Figure 2. Disjunctive invariants.

Then we arbitrarily choose one negative sample and all positive samples and continue to use linear kernel SVM. Of course, we cannot get a correct result, because positive samples are distributed among multiple clusters. Under these circumstances, positive and negative samples cannot be linear separated. The solution is that we use k-means++ to 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. There are some differences from the last example. 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 results are correct in the end.

3. Related Work

The closest related work for invariants generation were based on machine learning algorithms and guess-and-check model. In [24], the authors originally proposed a method using SVM to compute invariants. For linear invariants, when the samples could not be linearly separated, the authors come up with a method, called IOH, to get the invariants by using linear kernel SVM many times. In [17], the authors proposed to apply IOH to synthesize conjunctive invariants and path-sensitive classification to synthesize disjunctive invariants. Moreover, the technique in that paper reduced the number of guess-and-check iterations by selective samples. In [15], the authors considered the problem of inferring the inductive invariants for verifying program safety as binary classification. They utilized decision tree algorithm to learn candidate invariants in form of arbitrary boolean combinations of numerical in-equalities. The paper [6] proposed a robust framework ICE for learning invariants. ICE had two components: a white-box teacher and a black-box leaner. The leaner synthesized invariants and the teacher checked the correctness of the invariants in each round. If the check failed, the teacher come up with constraints for leaner to refine invariants. This method was a typical form of guess-and-check model. The paper [23] described a general framework for computing invariants by iteratively executing two phases. The search phase applied randomized search to discover candidate invariants and the validate phase applied checker to validate the correctness of the candidate invariants. The paper [22] proposed a data driven approach for generating algebraic polynomial loop invariants. Firstly, the method collected the value of the variables by executing a specific program. Secondly, it obtained a data matrix according to a template and upper bound d. Next, this method used linear algebra techniques to compute a candidate invariant. Finally, it checked the effectiveness of the candidate invariants. Our work is inspired by above paper. Our method uses machine algorithms to synthesize invariants, adopts guess-and-check model to ensure the correctness of the invariants and collects selective samples to reduce the number of iterations.

4. Preliminaries

4.1. Sampling

If we regard a program as a transition system, the states of program can be split in 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 cannot 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 that to distinguish these two states [7] (see Figure 3). If we can assure any state in the program belongs to Init and Reach, this program is correct.

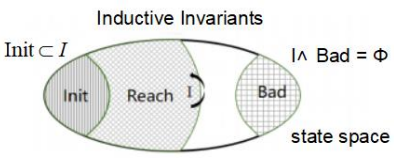


Figure 3. Program state space.

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 [20]. In other words, the initial value and their changed values which can make bad state is unreachable are both positive samples. Similar, bad states are 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.

In the following, we give the formalized description of good states and bad states. Suppose that we are given a Hoare triple in the following form:

```
{Pre}                /precondition/
While(Cond){C}          /Loopbody/
{Post}              /postcondition/
```

We assume that s represents those state before Pre, s' are the states in the loop body, and s'' are the states out of loop body but before the Post. The positive samples a negative samples are:

```
Positive states={
   s ∈ states,s' ∈ states,s'' ∈ states|
   s ∈ Pre ∧ s → s' → s''
   ∧ s'' ∉ Post
}
Negativestates={
   s ∈ states,s' ∉ states,s'' ∉ states|
   s ∉ Pre ∧ s → s' → s''
   ∧ s'' ∉ Post
}
```

4.2. K-means++

K-means++ proposed by Arthur and Vassilvitskii [1] in paper, is a kind of unsupervised machine learning algorithm on the basis of k-means for clustering. Given a train set and a hyper parameter k which decides the number of the result cluster, k-means is first to select k samples arbitrarily as the initial cluster center and then calculate the distance, Euclidean distance is the most common, from each sample point to each cluster center. Next, this algorithm classifies the nearest sample referred to the different cluster centers as one cluster. Afterwards, it recalculates the centroid of each sample cluster. Finally, it repeats the above steps until the centroid no longer changes. The difference between k-means++ and k-means is only the selection of the initial cluster center. k-means selects initial cluster center arbitrarily, but k-means++ selects initial cluster center according to the principle that the distribution of the initial cluster center is dispersed as far as possible. We refer the readers to [1] for details of the algorithm. This paper gets a conclusion that k-means++ improves both the speed and the accuracy of k-means through mathematical proof and experiments.

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 optimal value of the k by the way of iteration. The specific method will be discussed in section 4. It is because of this way, IOH is compatible with our algorithm, and also the most suitable value of k will also be found at the first time.

4.3. Algorithm for intersection for 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 cannot 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).

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

Algorithm 1: IOH

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

We realize IOH by the computer language of Python and apply it for the samples generated by make_circles function in the Python packages of sklearn. The result is showed in Figure 4. The circle points and triangle points represent negative samples and positive samples respectively. We can get the conclusion that IOH is feasible and inspiring.

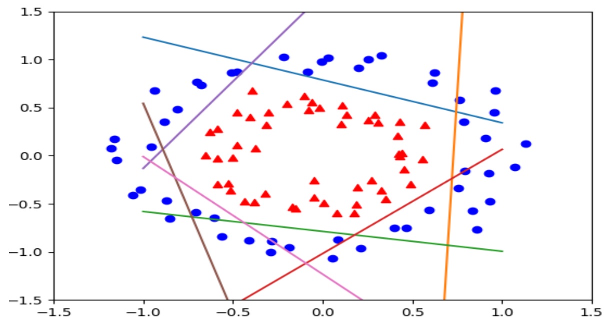


Figure 4. A inspiring result for IOH.

4.4. Hoare Logic

Hoare logic [22], also known as Floyd-Hoare logic, is a formal system developed by British computer scientist Hoare [13]. 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 precondition, *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 upon exit.

As the program with loop, the invariant is loop invariant. Loop invariant synthesis is a huge challenge [16]. It is an assertion that has been held since the beginning of the execution of the loop until the end of the loop. Just let a theorem prover execute only once, we can check whether invariant I is true. Here is the 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. According to Hoare logic, given an invariant I, by decomposing the formal description, we can get these two results:

$$\{I \wedge B\}C\{I\} \tag{3}$$

$$\{I \wedge !B\}\{Q\} \tag{4}$$

On the basis of existing theories, we can confirm that *I* is a correct invariant as long as $\{P\} => \{I\}$, $\{I \wedge B\}C\{I\}$ and $\{I \wedge !B\} => \{Q\}$. The sign of => means implicating. For example, x=*y* implicates x>=*y*. In other words, while the condition x=*y* holds, x>=*y* also holds.

In our experiments, the *I* in the program of Figure 1 is x=*y*, and P is x=*y*=0. $\{p\} => \{I\}$ 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 execution, and the program statement of error() is a true invariant.

If we found that I is an error invariant by theorem prover, we need to sample more. Specifically, if we just sampled 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 are integer, the invariant is equal to $x >= y$. $\{P\} => \{I\}$ holds, but $\{I \wedge !B\} => \{Q\}$ does not hold. It means some bad states are contained by I, so we should collect more samples and repeat our algorithm.

The above is a theoretical proof. In the actual experiment, Z3 theorem prover [21] helps us to verify the correctness of candidate invariants.

5. K++SVM

IOH is inspiring, but there are still problems waiting to be solved. Considering the program in Figure 2, 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, k++SVM emerges as the times required for solving this kind of problem. At first, we follow a program to sample positive samples and negative samples. Then, k++SVM 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 separate samples. If samples can be separated, we get the candidate invariants. Otherwise, SVM is used to eliminate single negative samples one by one for getting the expression of hyperplane equations. If it is valid, the candidate invariants are found. If not, this algorithm lets the value of k add 1 and cluster the positive samples again. Then repeat algorithm till it can distinguish samples 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 sample more states and repeat our algorithm.

Our algorithm is based on the guess-and-check model, some common problems with this model are that the effectiveness of algorithm is limited by the samples not only generated in the first phases [17], but also generated when the check fails. If the distribution of sampling points is reasonable, the iteration numbers between guess and check are fewer and the effectiveness of algorithm is higher. The most ideal situation is that our algorithm get the true invariants just only using guess-and-check model once. This situation is hard to achieve, but we have solutions to get close to this situation. Firstly, a large number of samples are needed, which can make sure that they include the samples characterizing the program. In fact, only the samples which can characterize the program are participated in the construction of invariants, because the support vector is only related to those samples. Secondly, we get the heuristics from paper [17]. According to this paper, sampling through verification provides useful new samples by paying a high cost. So it applies method selective Sampling() to selectively generate new sample while are closed to the candidate invariants. Our method also collect selective samples to improve the efficiency of algorithm.

The condition for the change of the value of k is that IOH cannot 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 M is not reduced. Every time, we choose one negative sample and then use SVM to separate the one negative sample from 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 M reduces, we continue to execute the algorithm. Otherwise, we stop IOH and let the value of k add 1. Here we give the pseudo code (see Algorithm 2).

Algorithm 2: k++SVM

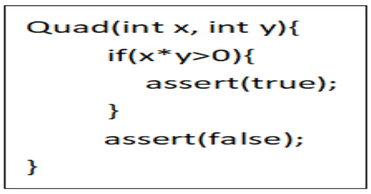
```
Input:   Positive sample:    X+
         Negative samples:  X-
         Cost parameter:c :=  1000
Output:  True Invariants: H
1:  k := 1
2:  Cluster(x+,k),obtain X: {X1+, X2+,...., Xk+}
3:  If (SVM can be used):
4:  |    For i=1 to k (i=k is allowed):
5:  |    |    h := Process (SVM (Xi+, {b}))
6: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

6. Implementation and Evaluation

We have implemented a prototype version of the algorithm described in this paper in the computer language of Python. First, we obtain constraint expression file whose type is txt by using CBMC tool (http://www.cprover.org/cbmc). CBMC is a Bounded Model Checker for 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 Satisfiability Modulo Theories (SMT) solver [21] for constraint solving. Next, to realize our algorithm and show result, numpy, sklearn, codecs, re and matplotlib packages of Python are adopted. Finally, we use SMT slover [21] to check whether the candidate invariants are true invariants.

The k-means++ and SVM in k++SVM are supported by off-the-shelf sklearn packages of Python. There is an important problem concerning the value that 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 Tables 1 and 2. In Table 1, Num is the experiment number. 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. There into, Fail means that finding invariants by IOH is failed; Our Invariants are the invariants output by using k++SVM; Total means the total number of the samples including positive samples and negative examples; K is the value of k in k++SVM; we can find the code of 13th experiment in Figure 5.



```
Quad(int x, int y){
    if(x*y>0){
        assert(true);
    }
    assert(false);
}
```

Figure 5. The code of 13th.

In Table 2, Times and our times is running time of the IOH and k++SVM respectively. The statistical time is from input samples to output candidate invariants.

Table 1. Experimental results.

NUM	File	LOC	Invariants	Our Invariants	Total	K	Type	Interproc
1	Figure 7 [10]	8	$x <= 8$	$x <= 8$	22	1	con	Y
2	Figure 1-a [24]	14	$(y > x - 0.5) \wedge (y < x + 0.5)$	$(y > x - 0.5) \wedge (y < x + 0.5)$	40	1	con	Y
3	ex1 [13]	22	$xa + 2ya > 2$	$xa + 2ya > 2$	36	1	con	N
4	Figure 2 [9]	18	$3x >= y$	$3x >= y$	62	1	con	Y
5	fse06 [10]	8	$(y > -0.5) \wedge (x > -0.5)$	$(y > -0.5) \wedge (x > -0.5)$	36	1	con	N
6	Figure 1[12]	6	$x - n <= 0$	$x - n <= 0$	43	1	con	Y
7	Figure 1-a [17]	8	$x <= y + 16$	$x <= y + 16$	100	1	con	Y
8	Figure 1-b [17]	9	Fail	$x > 0 \vee y > 0$	82	2	dis	N
9	Figure 1-d [17]	6	Fail	$x < 0 \vee 3y >= x + 1$	120	2	dis	N
10	Figure 1 [15]	10	Fail	$(y > -x + 0.5) \vee (y < -x - 0.5)$	50	2	dis	N
11	Figure 4 [11]	8	Fail	$(i < 0.5) \vee (j < i + 0.5)$	40	2	dis	N
12	Figure 1 [11]	8	Fail	$(y > 0.5) \vee (x < -0.5)$	42	2	dis	N
13	Quad	6	Fail	$(x > 0.5 \wedge y > 0.5) \vee (x < -0.5 \wedge y < -0.5)$	46	2	dis	N

Table 2. Running time comparison.

NUM	1	2	3	4	5	6	7	8	9	10	11	12	13
File	Figure 7	Figure 1-a	ex1	Figure 2	fse06	Figure 1	Figure 1-a	Figure 1-b	Figure 1-d	Figure 7	Figure 4	Figure 1	Quad
Times(ms)	1.946	1.578	2.529	3.494	1.989	5.550	5.405	Fail	Fail	Fail	Fail	Fail	Fail
OurTimes(ms)	1.927	1.627	2.488	4.559	2.594	6.933	5.940	23.630	33.490	43.865	3.128	159.152	30.456

In the last column of Table 1, Interproc means whether the true invariants can be synthesized by the verifier Interproc. Y and N respectively indicate that it can or cannot synthesize true invariants. Interproc [14] is an interprocedural analyzer for a small imperative language with procedure calls. In the basis of the technique of abstract interpretation, it infers invariants on the numerical variables of analyzed program.

The invariant comparison of experimental results are shown in Table 3. Num is the experiment number in Table 1; Reference means the source of the code; Invariants represent the results given in source; Our Invariants are the results obtained by our method.

Table 3. Invariants comparison.

NUM	**Reference**	**Invariants**	**Our Invariants**
2	[24]	$x = y$	$(y > x - 0.5) \wedge (y < x + 0.5)$
5	[28]	$(y >= 0) \wedge (x >= 0)$	$(y > -0.5) \wedge (x > -0.5)$
10	[15]	$x + y \neq 0$	$(y > -x + 0.5) \vee (y < -x - 0.5)$
12	[11]	$(x <= -1) \vee (y >= 1)$	$(x < -0.5) \vee (y > 0.5)$

Comparing our experimental results with other paper, there is nothing different between them (see Table 3). For example, in our 12th experiment, the invariants are $(x < -0.5) \vee (y > 0.5)$, but the result in paper [11] is $(x <= -1) \vee (y >= 1)$. In our 5th experiment, the invariants are $(y > -0.5) \wedge (x > -0.5)$, but in paper [4] the invariants are $(y >= 0) \wedge (x >= 0)$. The reason is that the type of the variables is integer, and the hyperplane parameter exported by SVM are the type of float. Just change our results to 0.5 units, we can get the same results. More specifically, in our 2rd experiment, by simplifying our result, we can get the conclusion $(y >= x) \wedge (y <= x)$. As the final result, our invariant is x = y which is the same as the result in paper [24].

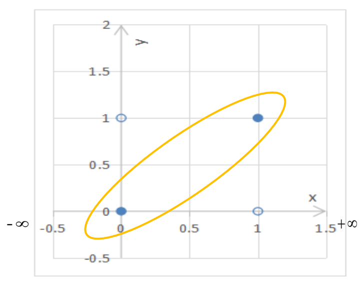


Figure 6. The code of 9th experiment.

In section 3.4, we prove that x=y is the true invariant. Suppose we use other machine learning algorithms, rather than linear kernel SVM, we get a nonlinear invariant (the form of circle in Figure 6). We also can prove that this kind of result is wrong. This gives us a heuristic idea. For a specific program, we are not supposed to only pursue that the algorithms can distinguish positive and negative samples, in spite of the essential characteristics of the change of the program variables. This point is well verified why the form of invariants should be linear.

In order to evaluate our method in synthesizing invariants, we investigate the following research question:

- RQ1 the first research question which we would like to answer to is: does our method help to synthesize disjunctive invariants?

Through the 8~13th experimental results in Table 1, it is obvious that the invariants with the form of disjunctive can only be computed by our method, while the IOH and Interproc cannot. Compared with Interproc, it is because that our method regards the problem of synthesizing invariants as the problem of classification by sampling and using classification algorithm. By analyzing samples rather than specific procedural logic, our method can synthesize disjunctive invariants. Not like the IOH, our method should cluster the positive samples before classification. According to our results, whether the target invariants are conjunctive or disjunctive depend on the number of the positive sample' clusters. If the number of positive sample' cluster, that is the value of k, is one, the target invariants are conjunctive. Otherwise, they are disjunctive. However, the number of the positive sample' clusters are not determined by the distance, but, intrinsically, the situation that whether the positive samples clusters can be separated clearly from negative samples. This point is also just reflected in the condition that the value of k should be changed when the samples cannot be separated clearly.

- RQ2 the second research question which we would like to answer is: does our method have the ability to automatically synthesize invariants? IOH and Interproc cannot synthesize disjunctive invariants and Interproc cannot compute the invariants in 3th and 5th experiment. What's more, when we using Interproc, we should manually choose the form of target invariant first. Specifically, only when we choose convex polyhedra option to complete our 4th experiment and choose octagon option to complete 2th experiment, Interproc is able to get the right invariants. This is a contradictory problem. In fact, before executing Interproc, we don't know which option should be chosen to get the right results. Through our experimental results, only our method can automatically synthesize disjunctive invariants.

Because our method obtains the program feature by the relation of samples, rather than the logic relation in a program. And then our method judges that the invariants are conjunctive or disjunctive by the value of k. If k is equal to 1, the invariants are conjunctive. Otherwise, they are disjunctive. The way to determine the value of k is that we gradually increase the value of k by 1 with the condition the positive samples cannot be separated clearly from negative samples. The optimal value of the k can be founded by the way of iterating. That is just the reason why our method have ability to automatically synthesize invariants.

- RQ3 the third research question is: how about the performance between IOH and our method? As the disjunctive invariant, IOH is useless. As the conjunctive invariant, referring to the Table 2, the running time of k++SVM is closed to IOH. Generally speaking, the running time of k++SVM is slightly higher than IOH, but sometimes slightly lower. It is reasonable that k++SVM is slightly higher, because k++SVM is more complex and should use k-means++ algorithm extra. The situation that k++SVM is slightly lower is unexpected, but it is still reasonable, because both the k++SVM and IOH are defective. When they use SVM, they choose one negative sample randomly. This randomness results that the iterations of SVM cannot be controlled, although it does not have a great impact on the results. This is the point that we will research in the future.
- RQ4 the fourth research question is: how about the performance in synthesizing disjunctive invariants by our method? Generally, referring to the Table 2, the running time of synthesizing disjunctive invariants is much higher than the running time of synthesizing conjunctive invariants by our method. It is reasonable, because the method of synthesizing disjunctive invariants have repeatedly used the method of synthesizing conjunctive invariants. In any case, because the time unit is still in the millisecond level, the running time of synthesizing disjunctive invariants by our method is acceptable.

7. Conclusions

We have shown that synthesizing invariants by machine learning algorithm can be profitable. In particular, when the target invariants are disjunctive, the k++SVM based on k-means++ and SVM is valid. 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++SVM is not only compatible with IOH, but can synthesize disjunctive invariants that IOH and Interproc cannot solve.

As the future work, we would like to modify our algorithm to eliminate the randomness and to make it more stable and robust.

References

[1] Arthur D. and Vassilvitskii S., “K-Means++: The Advantages of Careful Seeding,” *in Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, New Orleans, pp. 1027-1035, 2007.

[2] Cortes C. and Vapnik V., “Support-Vector Networks,” *Machine Learning*, vol. 20, no. 3, pp. 273-297, 1995.

[3] Cousot P. and Cousot R., “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints,” *in Proceedings of the* ACM *Sigact-Sigplan Symposium on Principles of Programming Languages*, New York, pp. 238-252, 1977.

[4] D'Silva V., Kroening D., and Weissenbacher G., “A Survey of Automated Techniques for Formal Software Verification,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165-1178, 2008.

[5] Ding Z., Wang R., Hu J., and Liu Y., “Detecting Bugs of Concurrent Programs with Program Invariants,” *in Proceedings of the IEEE International Conference on Software Quality, Reliability and Security Companion*, Vienna, pp. 1-15, 2016.

[6] Garg P., Löding C., Madhusudan P., and Neider D., “ICE: A Robust Framework for Learning Invariants,” *in Proceedings of the International Conference on Computer Aided Verification*, Vienna, pp. 69-87, 2014.

[7] Garg P., “Learning-Based Inductive Invariant Synthesis,” Thesis, Univerisity of Illinois Urbana-Champaign, 2015.

[8] Garg P., Neider D., Madhusudan P., and Roth D., “Learning Invariants Using Decision Trees and Implication Counterexamples,” *Acm Sigplan Notices*, vol. 51, no. 1, pp. 499-512, 2015.

[9] Gulavani B., Chakraborty S., Nori A., and Rajamani S., “Automatically Refining Abstract Interpreta-tions,” *in Proceedings of the Theory and Practice of Software, International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems*, Budapest, pp. 443-458, 2008.

[10] Gulavani B., Henzinger T., Kannan Y., and Nori A., “SYNERGY: A New Algorithm for Property Checking,” *in Proceedings of the ACM Sigsoft Inter-national Symposium on Foundations of Software Engineering*, Portland, pp. 117-127, 2006.

[11] Gulwani S., Srivastava S., and Venkatesan R., "Program Analysis as Constraint Solving," *in Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation*, Tucson, pp. 281-292, 2008.

[12] Gupta A. and Rybalchenko A., "InvGen: An Efficient Invariant Generator," *in Proceedings of the International Conference on Computer Aided Verification*, Grenoble, pp. 634-640, 2009.

[13] Hoare C., "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, no. 1, pp. 53-56, 1969.

[14] Jeannet B., "Interproc Analyzer for Recursive Programs with Numerical Variables," http://popart.inrialpes.fr/interproc/interprocweb.cgi, pp. 6-11, Last Visited, 2010.

[15] Krishna S., Puhrsch C., and Wies T., "Learning Invariants using Decision Trees," *Computer Science*, vol. 21, no. 7, pp. 44-59, 2015.

[16] Lin S., Sun J., Xiao H., Liu Y., Sanán D., and Hansen H., "FiB: Squeezing Loop Invariants by Interpolation between Forward/Backward Bredicate Transformers," *in Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, Urbana, pp. 793-803, 2017.

[17] Li J., Sun J., Li L., and Le Q., "Automatic Loop-Invariant Generation and Refinement through Selective Sampling," *in Proceedings of the Ieee/Acm International Conference on Automated Software Engineering*, Buenos Aires, pp. 782-792, 2017.

[18] Jun S., Pham L., Thi L., Wang J., and Peng X., "Learning Likely Invariants to Explain Why a Program Fails," *in Proceedings of the International Conference on Engineering of Complex Computer Systems*, Melbourne, pp. 70-79, 2018.

[19] Mcdonald J., Trigg T., Roberts C., and Darden B., "Security in Agile Development: Pedagogic Lessons from an Undergraduate Software Engineering Case Study," *in Proceedings of Cyber Security Symposium*, Coeur d'Alene, pp. 127-141, 2015.

[20] Mcmillan K., "Interpolation and SAT-Based Model Checking," *in Proceedings of the Computer Aided Verification, Berlin Heidelberg*, Germany, pp. 1-13, 2003.

[21] Moura L. and Bjørner N., "Z3: An Efficient SMT Solver," *in Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Budapest, pp. 337-340, 2008.

[22] Sharma R., Gupta S., Hariharan B., Aiken A., Liang P., and Nori A., "A Data Driven Approach for Algebraic Loop Invariants," *in Proceedings of European Conference on Programming Languages and Systems*, Rome, pp. 574-592, 2013.

[23] Sharma R., "From Invariant Checking to Invariant Inference Using Randomized Search," *in Proceedings of Computer Aided Verification*, Vienna, pp. 88-105, 2014.

[24] Sharma R., "Interpolants as Classifiers," *in Proceedings of International Conference on Computer Aided Verification*, Berlin, pp. 71-87, 2012.

[25] Sharma R. and Aiken A., "Verification as Learning Geometric Concepts," *in Proceedings of International Static Analysis Symposium*, Seattle, pp. 388-411, 2013.

[26] Stavnycha M., Yin H., and Römer T., "A Large-Scale Survey on the Effects of Selected Development Practices on Software Correctness," *in Proceedings of International Conference on Software and System Process*, Korea, pp. 117-121, 2015.

[27] Uqaili I. and Ahsan S., "Machine Learning Based Prediction of Complex Bugs in Source Code," *The International Arab Journal of Information Technology*, vol. 17, no. 1, pp. 26-37, 2020.

[28] Vizel Y., Gurfinkel A., Shoham S., and Malik S., "IC3 - Flipping the E in ICE," *in Proceedings of International Conference on Verification, Model Checking and Abstract Interpretation*, Paris, pp. 521-538, 2017.

Shengbing Ren was born on August 4, 1969 at Yueyang City, Hunan Province, China. He received his Bachelor of Science degree in Computer Software from Department of Mathematics, Huazhong Normal University, Hubei Province, China in 1992. He received his Master's degree in Computer Application Technology in 1995 from Department of Computer, Central South University of Technology, Hunan Province, China. He received his Doctor's degree in Control Theory and Control Engineering in 2007 from School of Information Science and Engineering, Central South University, Hunan Province, China. His research interests include: software engineering, embedded system, image processing, pattern recognition, dependable software. He is a professor in School of Computer Science and Engineering, Central South University, China. He is a Senior Member of China Computer Federation. He accomplished 11 research projects including 2 the National Natural Science Foundation of China as a key member, and over 60 papers was published. Now, he is dedicated to the research concentrated mostly on dependable software and pattern recognition.

Xiang Zhang was born on April19, 1993 at Anqing City, Anhui Province, China. He received his Bachelor of Science degree in communication engineering, Central South University of forestry science and technology University, Hunan Province, China in 2015. He received his Master's degree in software engineering, Central South University of Technology, Hunan Province, Chinain 2019.His research interests include: software engineering, machine learning. He is a student in School of Computer Science and Engineering, Central South University, China.