A Fast Invariant Inference Framework by Active Learning and SVMs

Li Jiaying[1], Sun Jun[2]
Singapore University of Technology and Design
[1]jiaying_li@mymail.sutd.edu.sg [2]sunjun@sutd.edu.sg

***Abstract*—We introduce a fast invariant inference framework based on active learning and SVMs (Support Vector Machines) which can learn most invariant of arithmetic form and also shorten the learning time greatly than before. Given a program containing loops along with precise pre-conditions and post-conditions, our approach can learn an adequate invariant fast to help program verification or provide an counter-example to assist the programmer to improve their programs. By invoking two phases iteratively: learning phase with the idea of active learning and SVMs (with or without kernels), and checking phase with the help of KLEE, a symbolic execution tool which can enumerate all the possible program paths, our implementation shows, compared with other off-the-shelf tools, this approach is quite an effective and efficient option for invariant inference task.**

***Index Terms*—invariant inference, active learning, SVMs, symbolic execution.**

I. Introduction

SOFTWARE correctness plays quite a very crucial role in today's digital and program-driven world. Among all of traditional program correctness checking approaches, discovering adequate inductive invariants is at the heart of automated program verification. As synthesizing invariants is also the hardest problem in this field, many computer scientists have tried many ways to solve this, such as template-based approachesinterpolationcounter-example guided predicate abstractionapplying machine learning techniquesetc. Once we get adequate inductive invariants, program verification can be easily solved by state-of-the-art solvers automatically.

A. *Static approach v.s. Dynamic approach*

In general, current approaches to find invariants can be grouped into two categories: static approaches which synthesize invariants based on program source code without running program, and dynamic approaches which produce invariants based on program runs.

For static approaches, the good news is it can generate very precise but complex invariants in principle, which always sacrifice easy generalization in return. And the effectiveness of static approaches really depends on the complexity of code. For some complicated code fragment, synthesizing invariants by static approaches is not only impratical but impossible.

On the contrast, dynamic approaches can be completely agnostic of the program. They may come up with simple but well-generalized invariants as they are simply constrained to learn some predicate that is consistent with given program runs. What's more, machine learning or data mining, [] which has advanced so quickly oven these years, offers quite a lot of techinical alternatives to assist solving invariant learning problem. [1] and [2] even try to solve this problem using decision tree. As a result, dynamic approaches to learning invariants have gained popularity in recent years.

In dynamic approaches such as [3] and [4], they often split the synthesizer of invariants into two parts: a honest teacher and a learner. And the learning procedure can be divided into many rounds of "guess and check": in guess phase, the learner learn from given samples and propose an invariant hypothesis $\mathcal{H}$ to the teacher, then the teacher check whether the hypotheis is adequate to verify the program and send some feedback to the learner in check phase. The process ends until the teacher agrees on learner's hypothesis.

B. *Passive learning v.s. Active learning*

In one of our referenced papers [4], the author shows ICE-learning algorithm can guarantee convergence on a finite class $\mathcal{C}$ of concepts. However it does not state how fast this learning process can come to convergence. Actually, in the worst case it can get the right classifier until it has tried all the other wrong classifers, which means the learning converge very slowly under this condition. From our understanding, this slow convergence problem is due to the passive learning nature of ICE-learning. In our framework, we try to develop an active learning framework to get rid of this speed problem mainly through two means: states chains, which will be present in next subsection; and active learning, in which the learner can have more opportunities to interact with the teacher to refine its hypothesis.

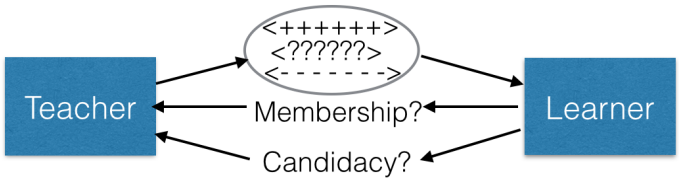


Fig. 1. Fast Invariants Inference Framework

We get this inspiration from L^*- *learning algorithm* []. When the learner try to learn a consistent DFA, it can ask two kinds of queries: membership queries and candidacy queries. The aim of membership queries is to check whether a certain sample satisfies the real model or not; candidancy query is

in order to decide whether the guess model is idenetical with the real model. Return back to ICE-learning [4], in which the learner can only ask the candidacy query about whether a hypothesis invariant is really the expected invariant or not. The bad thing is, in order to refute a invariant, the teacher has to check with constraint solver or some other time-consuming tools. What if we can refute a hypothesis by just one or two program runs?

In our framework, along with candidacy query, our learner can ask membership query as shown in Fig. 1, which makes most of previous time-consuming validation job reduce to time-saving task of running test cases. As a result, our framework can correct wrong guesses much easier and earlier, and moreover, this makes our learning framwork defeat other approaches in speed.

C. State Chain

A typical program with one loop expresion annotated with precise pre-condition and post-condition can be formalized as following form:

$$assume\ P;\ while\ B\ do\ S\ od;\ assert\ Q$$

The loop has a pre-condition P, which should be satisfied before entering the loop expression. Predicate B guards the loop entry which is the only way towards the loop body S. Any states comming satify pre-condition should satisfy also post-condition Q after execution of the loop. Given a loop invariant $\mathcal{I}$, we can prove that the assertion holds if the following three properities are valid:

$$P \Rightarrow \mathcal{I} \tag{1}$$

$$\{\mathcal{I} \wedge B\} S \{\mathcal{I}\} \tag{2}$$

$$\mathcal{I} \wedge \neg B \Rightarrow Q \tag{3}$$

Good State, Bad State & Implication: These three concepts are introduced in [3]. Let $\mathcal{C}$ be a candidate invariant.

From equation (1) we know, for any invariant $\mathcal{I}$, any state that satisfies P also satisfies $\mathcal{I}$. We call any state that must be satisfied by an actual invariant a good state.

Now consider equation (2). A pair (s, t) satisfies the property that s satisfies B and if the execution of S is started in state s then S can terminate in state t. Since an actual invariant I is inductive, it should satisfy $\mathcal{I}(s) \Rightarrow \mathcal{I}(t)$. Hence, a pair (s, t) satisfying $\mathcal{C}(s) \wedge \neg\mathcal{C}(t)$ proves $\mathcal{C}$ is not an invariant.

Finally, consider equation (3). A satisfying assignment b of $\mathcal{C} \wedge \neg B \wedge \neg Q$ proves $\mathcal{C}$ is inadequate to discharge the post-condition. For an adequate invariant $\mathcal{I}$, $\mathcal{I}(b)$ should be false. We call a state that must not be satisfied by an adequate invariant, such as b, a bad state.

Good State Chain, Bad State Chain & Implication Chain: In our approach, we assume $\{s_0, s_1, s_2, ..., s_i, ..., s_n\}$ is a chain of states in target program, where s_0 is the initial state before entering the loop, and s_i is a state just after the loop has iterated i times in program. And we assume s_n is the state which satisfy $\neg B$ so it is the state that can jump out the loop body.

For a state chain $\{s_0, s_1, s_2, ..., s_i, ..., s_n\}$, if s_0 satisfies P, and s_n satifies Q, we say this is a good state chain, Because if state s_0 satisfy P, s_0 is good state that must satisfy I, according to equation 1. And what's more, according to equation 2, all of $\{s_1, s_2, ..., s_i, ..., s_n\}$ are good states. So now we can get a good state chain $\{s_0, s_1, s_2, ..., s_i, ..., s_n\}$.

On the contrary, for a state chain $\{s_0, s_1, s_2, ..., s_i, ..., s_n\}$, if s_0 satisfies $\neg P$, and s_n satifies $\neg Q$, we say this is a bad state chain, which means all the states in this chain belong to bad state set.

Actually for an arbatery state chain there are also two other possibilities we have not mentioned yet. One is a chain begin with a state s_0 statify P but end with a state s_n satisfy $\neg Q$, this is a counterexample to disprove the program. If we come to this case, it means there is something wrong with at least one of pre-condition, loop condition, loop body or post-condition. We need to find out what happens and update the program, after which we can reapply our approach to learn loop invariants. The other case is a chain begin with a state s_0 statify $\neg P$ but end with a state s_n satisfy Q. Under this condition, we could not justify whether s_0 and s_n satisfy invariants or not, not to mention the other states $\{s_1, s_2, ..., s_i, ..., s_{n-1}\}$. The only thing we can ensure is this is an implication chain. In total, we can have table. I.

TABLE I
STATE CHAIN - INVARIANT TABLE

$\{s_0, s_1, ..., s_n\}$	$s_n \models Q$	$s_n \models \neg Q$
$s_o \models P$	*good state chain*	*counter example*
$s_0 \models \neg P$	*implication state chain*	*bad state chain*

Among all these possibilities, we can get as many as, if not more than, samples than previous approaches with running program just once. So with the sample information, the learner can learn an as good, even if not better, invariant than the previous approaches, as it can utilize more information to do invariant learning task. This also imply our approach can get convergence faster than before.

D. Active Learning

With these samples we can get from program runs, the learner can apply machine learning classify algorithm, for instance, SVMs, to divide samples apart. After the learner get a classifier out, we know the points along with the clasifier means most. Which also means, if we get a wrong classifier, it is very likely that the predict labels of points along with the classidier get wrong. So these points are most potential invalidation points We randomizely pick a few of these points to ask the teacher for membership query. Then the teacher run the target program with these points to label them based on our state chain therom. After returning to the learner these new samples back, the learner start to learn a classifier again. If the new learned classifier is identical with the previous one, we believe this learning process get converged. Otherwise, our approach repeat this procedure until classifier converges or the quantities of interaction reach a manually threhold we force to terminate.

E. Invariants Checking by Symbolic Execution

There is always a check phase after invariant guess phase. Many other approaches use a decision procedure to validate the given candidacy. However, there is a symbolic execution tool called KLEE available, which release our burden in checking work. In our implementation, we use KLEE as an invariant verifier to help us validate hypothesis invariants from the learner.

A simple KLEE program can be written as following:

```
#include "klee/klee.h"

int main(int argc, char** argv)
{
  int x, y;
  klee_make_symbolic(&x, sizeof(x), "x");
  klee_make_symbolic(&y, sizeof(y), "y");

  klee_assume(x + y > 0);
  x--;
  y++;
  klee_assert(x + y > 0);
}
```

Listing 1. KLEE example 1

After we compile the source file, we can use KLEE to do symbolic execution on the generated object file. KLEE will enumerate all the possible paths and then pass their path condition to a solver to get concrete values for all the symbolic variables. Note that, for any program ran by KLEE, if all the possible paths pass the assertion, we can ensure the correctness of the program. In other words, we have proved the correctness of the program.

Look back to this example, we can only get one path, and that path passes the asserion. Here we can also get a concrete input $(x, y) = (-1335664895, -811818755)$. (Note: there is no special meaning of these two numbers, but in this case x + y = 2147483646, there is an addition overflow)

So after the learning process in last subsection, the learner starts candidacy query with a good hypothesis invariant $\mathcal{H}$. In our approach, the teacher divide the target program into three loop-free parts to check each of the equations to check whether the hypothesis invariant $\mathcal{H}$ is an adequate inductive invariant. If not, the teacher can come up with an positive example, a negative example or a implication.

With these feedback, the learner can refine the learning hypothesis and then repeat the same procedure util the hypothesis can pass all the three equations which means we have found an adequate invariant to do verification or util it find out a counterexample which pass the pre-condition but fails at post-condition after execute loop.

F. Contributions and Future Work

The main contribution of this paper is to propose a new invariant inference framework based on active learning. It extends ICE-learning, but converge faster than the latter one. We exploit one program run to get a state chain rather than just one state, and as a result, our approach can get more precise result from same numbers of program runs. Our approach adapts active learning rather than passive learning. This adaption enable the learner to get a good enough hypothesis before submission, for the teacher can help the learner to focus on his fault as early as possible.

Another contribution is we using SVMs to do the learning. Although we can only learn linear invariants at this moment, we can get more complex arithmatic invariants after applying SVMs with kernel method (such as RBF kernel) in a comming future. The bad news is we do not have a powerful verifier which can reasoning about logistic funcion straightly, so we can not do the proof directly. We plan to use serveral linear inequtions to approximate the curve we get from SVMs. After finishing these, our framework can be powerful enough to get all the arithmetic invariants in theory.

II. An Example

This section will show you the exact steps how our framework works on a simple example. Consider the C program in the below. This program requires an invariant for its verification.

```
int x, y;
assume (x + y > 0);
while (x >= 0) {
  x--;
  y++;
}
assert (y >= 0);
```

Listing 2. loop example

A. Preprocessing

Before learning invariant, we need to prepare the program to be ready for recording data. In this preprocessing step, we apply a simple instrumentation to the target program source code in order to get some information from test runs. In this step, we output the value of variables in every loop iteration (which can be viewed as a state chain) and whether the first state satisfy the pre-condition and the last state satify the post-condition or not. After this step, our program become to the following:

```
int x, y;
if (x + y > 0) print ("pass");
else           print ("fail");

while (x >= 0) {
  print (x, y);
  x--;
  y++;
}
print (x, y);

if (y >= 0) print ("pass");
else        print ("fail");
```

Listing 3. loop example after instrumentation

B. Active learning using SVMs

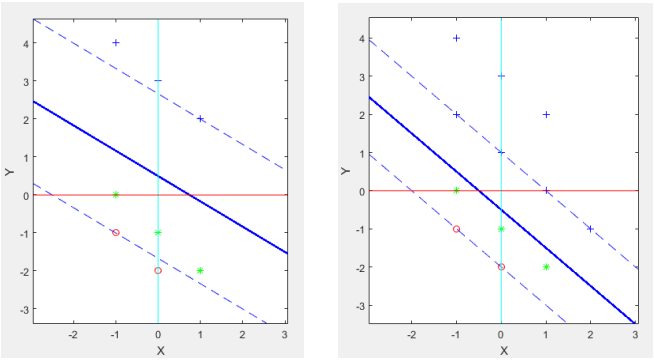
This is main step doing actual learning in our framework. In this step, the teacher run program with test cases generated randomly or membership queries from the learner. For instance, after running instrumented program with three randomized inputs $(x, y) = (1, 2), (1, -2), (0, -2)$. we can get the following three program state chain:

- pass (1, 2) (0, 3) (-1, 4) pass
- fail (1, -2) (0, -1) (-1, 0) pass
- fail (0, -2) (-1, -1) fail

According to Table. I, we can figure out which cases these three state chain belongs to. The results are listed in the following:

- [+] $(1, 2) \rightarrow (0, 3) \rightarrow (-1, 4)$
- [?] $(1, -2) \rightarrow (0, -1) \rightarrow (-1, 0)$
- [-] $(0, -2) \rightarrow (-1, -1)$

With these state chain, the learner can apply SVMs(without kernel) to learn a classifier as a hypothesis invariant. The learning hyperplane (with its margin) is shown in Fig. 2(a). In that figure, we label each kind of samples with different marks: sample in good state chain with blue plus mark, sample in bad state chain with red circle mark, and sample in implication state chain with green star mark.



(a) Before membership query (b) After membership query

After getting a hyperplane (here it is a line as shown in Fig. 2(a)) which separate the whole space into two half spaces:

$$\mathcal{H}_0 : 4x + 6y >= 3$$

the learner checks whether any implication state chain refute the hypothesis. Note that, there is only one case, in which there is a state with positive label predicted by hypothesis is followed by a state with negative label, that can refute hypothesis. After checking, as the hypothesis invariant passes this check in this example, the learner has got more confidence with his hypothesis.

Then the learner picks up some points on or near the invariant margin to invoke membership queries, two queries with $(x, y) = (0, 1), (2, 1)$ here for instance.

Having membership queries from the learner, the teacher run the program with input from query content, and can easily get the following outputs:

- pass (0, 1) (-1, 2) pass
- pass (2, -1) (1, 0) (0, -1) (-1, 2) pass

So now we have two good state chains.

- [+] $(0, 1) \rightarrow (-1, 2)$
- [+] $(2, -1) \rightarrow (1, 0) \rightarrow (0, -1) \rightarrow (-1, 2)$

with these new samples, the learner starts to learn the classifier again to find a better invariant hypothesis which is shown in Fig. 2(b):

$$\mathcal{H}_1 : x + y >= -1/2$$

The learner then come up with new membership queries and the same procedure repeats until the hypothesis stay the same or the iteration number come up to a manually set threshold. Finally, the learner know the hypothesis invariant he needs is just $\mathcal{H}_1$.

C. Validation by KLEE

As KLEE can not deal with double type, combining the observation that x and y are integer number, we convert the hypothesis into

$$\mathcal{H}_2 : x + y > -1$$

To validate the hypothesis $\mathcal{H}_2$, we separate the target program into three parts as following:

```
1  int x, y;
2
3  // Part 1: Before Loop
4  klee_assume (x + y > 0);
5  klee_assert (x + y > -1);
6
7  // Part 2: Loop Body
8  klee_assume (x >= 0);
9  klee_assume (x + y > -1);
10   x--;
11   y++;
12 klee_assert (x + y > -1);
13
14 // Part 3: After Loop
15 klee_assume (x < 0);
16 klee_assume (x + y > -1);
17 klee_assert (y >= 0);
```

Listing 4. Validation by KLEE

By running KLEE on these three loop-free programs, we find out the first two program with no assertion failure, but the third one fails when $(x, y) = (-2147483648, -1073741825)$.

We can find out this is a counterexample which satisfies pre-condition but will fail at post-condition. Hence, it becomes developers' task to fix this problem with the help from this counterexample sample.

In a nutshell, after preprocessing, learning and checking, our framework finds out an counterexample which can triger a bug in this illustrative example.

References

[1] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," 2015.

[2] S. Krishna, C. Puhrsch, and T. Wies, "Learning invariants using decision trees," *arXiv preprint arXiv:1501.04725*, 2015.

[3] R. Sharma and A. Aiken, "From invariant checking to invariant inference using randomized search," in *Computer Aided Verification*. Springer, 2014, pp. 88–105.

[4] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "Ice: A robust framework for learning invariants," in *Computer Aided Verification*. Springer, 2014, pp. 69–87.