An Invariant Inference Framework by Active Learning and SVMs

Li Jiaying
Singapore University of Technology and Design
lijiaying1989@gmail.com

***Abstract*—We introduce a fast invariant inference framework based on active learning and SVMs (Support Vector Machines) which aims to systematically generate a variety of loop invariants efficiently. Given a program containing a loop along with a precondition and a post-condition, our approach can learn an invariant which is sufficiently strong for program verification or otherwise provide a counter-example to assist the programmer to locate the bug. By invoking two phases iteratively: the learning phase with the idea of active learning and classification, and the checking phase with the help of a constraint solver, our preliminary experiment shows, compared with other off-the-shelf tools, this approach may be potentially more effective and efficient than existing approaches.**

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

I. Introduction

SOFTWARE correctness plays a crucial role in today's digital and software-driven world. For nearly all traditional program verification techniques, discovering loop invariants is at the heart of automated program verification. Many computer scientists have tried many ways to solve this, such as abstract interpretation[1], interpolation[2][3], counter-example guided predicate abstraction[4], applying machine learning techniques[5][6][7] etc. Once we get a loop invariant which is sufficiently strong, program verification can be easily solved by state-of-the-art solvers automatically.

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

In general, existing approaches on finding 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 executions.

For static approaches, the advantage is it can generate very precise but complex invariants in principle. The effectiveness of static approaches depends on the complexity of code. For some complicated code fragment, synthesizing invariants by static approaches is not only unpractical but impossible[8].

On the contrast, dynamic approaches can be completely agnostic of the program. They may come up with invariants efficiently as they are simply constrained to learn some predicate that is consistent with given program executions. What's more, machine learning[9] or data mining[10], which has advanced so quickly over these years, offers quite a lot of technical alternatives to assist solving invariant learning problem. For instance, in [6] and [7], the authors applied learning algorithms based on decision trees to learn loop invariants. As a result, dynamic approaches to learning invariants have gained popularity in recent years.

In dynamic approaches such as [11] and [5], they often split the synthesizer of invariants into two parts: an honest teacher and a learner. The learning procedure can be divided into many rounds of "guess and check": in the guessing phase, the learner learns from given samples and propose an invariant hypothesis $\mathcal{H}$ to the teacher, then the teacher checks whether $\mathcal{H}$ is adequate to verify the program and sends some feedback to the learner in check phase. The process continues until the teacher agrees on learner's hypothesis.

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

In[5], Garg. et al. suggest a new learning paradigm called ICE-learning for synthesizing invariants by learning frm examples, counter-examples, and implications. The authors prove ICE-learning algorithm can guarantee convergence on a finite class $\mathcal{C}$ of concepts. However, they do not show how fast this learning process converges. In the worst case, it can get the right classifier until it has tried all the other wrong classifiers, which means the learning converge very slowly in this case. 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 overcome this limitation mainly through two means: state chains, which will be presented in next subsection; and active learning, in which the learner can have more opportunities to interact with the teacher to refine its hypothesis.

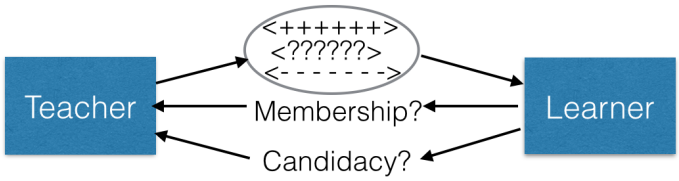


Fig. 1. Big View for Invariants Inference Framework

We observe in Anguluin's L^* *learning algorithm*[12], when the learner try to learn a consistent DFA, it can ask two kinds of queries: membership queries and candidate queries. The aim of membership queries is to check whether a certain sample satisfies the real model or not; a candidate query is in order to decide whether the learned model is identical to the real model. With respect to ICE-learning[5], in which the learner can only ask the candidate query about whether a candidate invariant is

really the expected invariant or not. The bad news is, in order to check whether the candidate invariant is indeed an invariant and is sufficiently strong for program verification, the teacher has to check through constraint solving or some equivalent techniques (which are often time consuming). What if we can refute a hypothesis by just several program executions?

In our framework, along with candidacy query, our learner can ask membership query as showed in Fig. 1, which makes most of the 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 framework defeat other approaches in speed.

C. State Chain

A typical program with one loop expression annotated with precise precondition and post-condition can be formalized in the following form:

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

The loop has a precondition P, which should be satisfied before entering the loop. Predicate B guards the loop entry which is the only way towards the loop body S. The goal is to prove that any state satisfying precondition 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 properties 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 [11]. In this paper, we use predicates and sets of states interchangeably. Let $\mathcal{C}$ be a candidate invariant.

From equation (1) we know, for an 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 $s \in \mathcal{I} \Rightarrow t \in \mathcal{I}$. Hence, a pair (s, t) satisfying $s \in \mathcal{C} \wedge t \notin \mathcal{C}$ proves $\mathcal{C}$ is not an invariant.

Finally, consider equation (3). The 'existence of a state $s \in \mathcal{C} \wedge \neg B \wedge \neg Q$ proves $\mathcal{C}$ is inadequate to discharge the post-condition. We call a state s which satisfies $/B \wedge\ /Q$ 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 the 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 the program. We assume s_n satisfies $\neg B$ 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 satisfies Q, we say this is a good state chain, Because if state s_0 satisfy P, s_0 is a good state that must satisfy $\mathcal{I}$, according to equation 1. Furthermore, 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 satisfies $\neg Q$, we say this is a bad state chain, which means all the states in this chain should be bad states.

Actually for an arbitrary state chain there are also two other possibilities we have not mentioned yet. One is a chain begins with a state s_0 that satisfies P but end with a state s_n that satisfies $\neg Q$, this is a counter-example to disprove the program. That means there is something wrong with at least one of precondition, 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 that satisfies $\neg P$ but end with a state s_n that satisfies Q. Under this condition, we could not justify whether s_0 and s_n satisfy invariants or not, not to mention 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 more samples than previous approaches can with executing program just once. So with the sample information, the learner can learn an as good invariant as, if not better than, the previous approaches, as it can utilize more information to do invariant learning task. This also implies 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 can compute the most informative state in order to improve the classifier Intuitively, if we get a wrong classifier, it is very likely that the predicted labels of points along with the classifier are wrong. So these points are most valuable. We randomly pick a few of these points to ask the teacher for a membership query. Then the teacher run the target program with these points to label them based on Table. I. 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 assume this learning process gets converged. Otherwise, our approach repeats the same procedure until classifier converges or until the number of interactions reaches the threshold we manually set to force to terminate.

E. Invariants Check by Symbolic Execution

There is always a checking phase after invariant guessing phase. Many existing approaches use a decision procedure

to validate the given candidacy. In our implementation, we use KLEE[13] as an invariant verifier to help us validate hypothesis invariants from the learner.

A simple KLEE program can be written as following:

```
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

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.

With regards to this example, we can only get one path which passes the assertion. And here ere 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, so 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 divides 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 a positive example, a negative example or an implication.

With these feedbacks, the learner decides to refine the hypothesis invariant or not. In the refining phase, the learner repeats the same procedure until the hypothesis passes all the three equations or the teacher finds out a counter-example which can disprove the program.

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 a more precise result from the same numbers of program executions. Our approach adapts active learning rather than passive learning. This enables the learner to get a good enough hypothesis before submitting a candidate query, as 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 arithmetic invariants after applying SVMs with kernel method (such as RBF kernel) in the future. The bad news is we do not have a powerful verifier which can reasoning of logistic function straightly, so we cannot do the proof directly. We plan to use several linear inequalities 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, xa, ya;
assume (xa + 2 * ya >= 0);
while (nondet()) {
  x = xa + 2 * ya;
y = -2 * xa + ya;
x++;
if (nondet()) y = y + x;
else y = y - x;

xa = x - 2 * y;
ya = 2 * x + y;
}
assert (xa + 2 * ya >= 0);
```

Listing 2. Loop Example

A. Preprocessing

Before learning invariant, we need to prepare the program to record 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 satisfies the precondition and the last state satisfies the post-condition or not. After this step, our program become to the following:

```
int x, y, xa, ya;
if (xa + 2 * ya >= 0) print (``pass'');
else  print (``fail'');

while (nondet()) {
print (xa, ya);
x = xa + 2 * ya;
y = -2 * xa + ya;
x++;
if (nondet()) y = y + x;
else y = y - x;

xa = x - 2 * y;
ya = 2 * x + y;
}
print (xa, ya);

if (xa + 2 * ya >= 0) print (``pass'');
else  print (``fail'');
```

Listing 3. Loop Example after Instrumentation

B. Active learning using SVMs

This is the main step in invariant inference in our framework. In this step, the teacher runs program with test cases from random generator or from membership queries. For example, after running the instrumented program with four randomized inputs $(x, y) = (9, -7), (7, 1), (-9, 12), (-15, 6)$.

we can get four program state chains, and figure out their labels according to Table. I. The results are listed in the following:

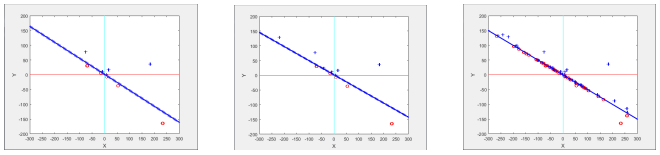
- fail (9, -7) (54, -37) (233, -164) fail [-]
- pass (7, 1) (16, 17) (183, 36) pass [+]
- pass (-9, 12) (-76, 78) (-217, 311) pass [+]
- fail (-15, 6) (-70, 30) (-367, 161) fail [-]

With these state chains, the learner can apply SVMs(without kernel) to learn a classifier as a hypothesis invariant. The learning hyper-plane (with its margin) is shown in Fig. 2(a).

$$\mathcal{H}_1 : 0.156591 * xa + 0.289092 * ya >= -0.385232$$

Then the learner checks whether any implication chain can refute the hypothesis. As there is not any implication chains here, we ignore this checking step.

Then the learner picks up some points on or near the invariant margin to invoke a new membership query, for instance, $(x, y) = (-41, 24)$.



(a) $\mathcal{H}_1$ (b) $\mathcal{H}_2$ (c) $\mathcal{H}_{15}$

Fig. 2. Refining Visualization

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

- pass (-41, 24) (-220, 130) (-1017, 611) pass [+]

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):

$$\mathcal{H}_2 : 0.172869 * xa + 0.360811 * ya >= -0.570898$$

As this process continues, the learner gains more and more confidence with his hypothesis.

The learner then comes up with new membership queries and the same procedure repeats until the hypothesis stays the same or the iteration number exceeds the threshold we set to force termination[1].

Finally, the learner gets the hypothesis invariant he needs as showed in Fig. 2(c) :

$$\mathcal{H}_{15} : 0.999835 * xa + 1.999670 * ya >= -1.000000$$

C. Validation by KLEE

As KLEE cannot deal with data of double type, combining the observation that x and y are integer number, we round-off the coefficients and convert the hypothesis into

$$\mathcal{H}_c : xa + 2 * ya >= -1$$

To validate $\mathcal{H}_c$, we separate the target program into three parts:

[1]You can find out how this learning converges dynamically by running the demo matlab script in our project folder, in case you are interested.

```
1  klee_make_symbolic(&xa, sizeof(xa), ''xa'');
2  klee_make_symbolic(&ya, sizeof(ya), ''ya'');
3
4  // Part 1: Before Loop
5  klee_assume (xa + 2 * ya >= 0);
6  klee_assert (xa + 2 * ya >= -1);
7
8  // Part 2: Loop Body
9  klee_assume (xa + 2 * ya >= -1);
10 // put loop bady here ...
11 klee_assert (xa + 2 * ya >= -1);
12
13 // Part 3: After Loop
14 klee_assume (xa + 2 * ya >= -1);
15 klee_assume (xa + 2 * ya >= 0);
```

Listing 4. Validation by KLEE

By running KLEE on these three loop-free programs, we find out the first program with no assertion failure, but the second part fails when $(xa, ya) = (-198640688, 492751876)$.

After executing the program with this initial values, we find out this is a counter-example: it satisfies precondition; but after executing the loop body once(assuming it chooses the else-branch in if-expression, fails at loop condition and thus jumps out of the loop, $(xa, ya) = (580522691, 1676896317)$, which fails at post-condition (due to integer addition overflow). Hence, it becomes developers' job to work to locate the bug and then fix it with this counter-example's help.

In a nutshell, after the preprocessing step, the learning phase and the checking phase, our framework finds out a counter-example which can trigger a bug in this illustrative example. After fixing this bug, we can apply our framework to prove program again or find a new bug.

REFERENCES

[1] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1977, pp. 238–252.

[2] R. Sharma, A. V. Nori, and A. Aiken, "Interpolants as classifiers," in *Computer Aided Verification*. Springer, 2012, pp. 71–87.

[3] A. Albarghouthi and K. L. McMillan, "Beautiful interpolants," in *Computer Aided Verification*. Springer, 2013, pp. 313–329.

[4] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer aided verification*. Springer, 2000, pp. 154–169.

[5] 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.

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

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

[8] C. A. Furia, B. Meyer, and S. Velder, "Loop invariants: Analysis, classification, and examples," *ACM Computing Surveys (CSUR)*, vol. 46, no. 3, p. 34, 2014.

[9] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[10] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

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

[12] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.

[13] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.