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 one 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 counter-examples to assist software developers to locate program bugs. By invoking learning and checking phases iteratively, our preliminary experiments show, this approach may be potentially more effective and efficient when compared with other existing approaches.**

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

I. Introduction

SOFTWARE correctness plays a crucial role in this digital and software-driven world. For most traditional program verification techniques, discovering loop invariants is at the heart of automated program verification. Once we get loops invariants which are sufficiently strong, program verification can be easily solved by state-of-the-art solvers automatically. To solve the invariant problem, top computer scientists have created a variety of approaches or even borrowed ideas from other fields, such as abstract interpretation[1], interpolation[2][3], counter-example guided predicate abstraction[4], applying machine learning techniques[5][6][7] etc. Despite of these innovative ideas, there is still a long way to go towards invariants inference in arbitrary program containing loops.

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 they can generate very precise but complex invariants in principle. But effectiveness of static approaches depends largely on the complexity of code. For complicated code which are usually written by developers, synthesizing invariants by static approaches is not only unpractical but impossible[8].

On the contrast, dynamic approaches make effort on a totally different direction, which 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] and data mining[10], which have been advanced so quickly over these years, offer quite a variety 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 learning-based dynamic approaches, such as [11] and [5], they often split the synthesizer of invariants into two roles: an honest teacher and a learner. And also split the learning procedure into many rounds of "guess and check": in the guessing phase, the learner learns from given samples and propose an invariant hypothesis $\mathcal{H}$, then in the checking phase, the teacher checks whether $\mathcal{H}$ is adequate to verify the program and sends some feedback to the learner . 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 from 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, the algorithm can get the right classifier until it has tried all wrong classifiers, which means the learning can converge very slow. This result is not casual because of the passive learning nature of ICE-learning. In passive learning, the learner can only digest given materials in hand without discovering more. In our framework, we develop an active learning framework to overcome this convergence limitation through two means: state chains, which provides the learner with more data and dependency relations in certain program executions; and active learning, in which setting the learner can have more opportunities to interact with the teacher to gain more learning materials to help refine its hypothesis.

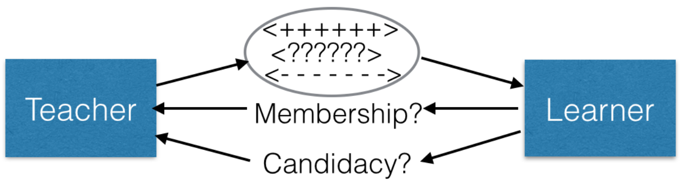


Fig. 1. Big View for Invariants Inference Framework

We get this intuition from 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. A membership query 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 consistent with the real model or not. With respect to ICE-learning[5], the learner can only ask the candidate queries without membership queries, which makes it hard and time-consuming to accept a model or reject one. Actually, in order to perform candidate check, the teacher has to call constraint solvers or equivalent techniques each time. What if we can ask membership queries?

We integrate this active learning idea into our framework as shown in Fig. 1. The learner can ask membership queries along with candidacy queries, which makes most of the previous time-consuming validation jobs reduce to several time-saving tasks of program executions. As a result, the teacher can reject wrong guesses easier and earlier, which makes each "guess and check" round more slim, and therefore our learning framework can defeat other approaches in speed.

C. State Chain

A typical program with one loop expression annotated with precondition and post-condition can be formalized as:

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

The program 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: [11] first instroduces these three concepts. 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, according to equation 1, s_0 is a good state that must satisfy $\mathcal{I}$. Furthermore, according to equation 2, any state following s_0 is a good state.

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, any state in which should be a bad state. Because if one of them is a good state, according to equation 2, all the states behind it should be good too. But the fact is the last state is a bad one. So this should not happen, and thus all states in bad state chain are bad.

There are still two other tricky possibilities for an arbitrary state chain we have not mentioned yet. One is a chain begins with a state s_0 that satisfies P but ends with a state s_n that satisfies $\neg Q$. It means there exists some inputs which can pass the precondition but fails at the post-condition, which are counter-examples to disprove the program. Then the developers need to find out what bug causes this failure and update the program, after which we can reapply our approach to learn loop invariants. The other case is a chain begins with a state s_0 that satisfies $\neg P$ but ends 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, which means if one state in the chain is a good state, the states after it are too; but if one state is a bad state, the state after it can either good or bad.

So 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*

In the previous approaches, the learner can get one good state, one bad state or one implication pair in one program execution. But in our framework, as shown in table. I, we can get one state chain in the same program execution, which offer the learner more samples to learn from. As a result, with the sample information, the learner can learn an as good invariant as, if not better than, the previous approaches. This also implies our approach can converge faster than before.

D. Active Learning

With samples we get from program executions and inferred labels information added, the learner can apply classification algorithms from machine learning area, for instance, SVMs, to divide samples with different labels apart. After getting a classifier out, the learner can compute the most informative samples in order to improve the classifier Intuitively, if we get a wrong classifier, it is very likely that the predicted labels of samples along with the classifier are wrong. So these points are most valuable. The learner then invoke membership queries to the teacher with some of these most informative points. 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, the learner starts 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 the number of interactions reaches a threshold for termination.

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 follows:

```
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 to object file, we use KLEE to perform symbolic execution on it. KLEE will enumerate all the possible paths and emit concrete values for all the symbolic variables to each path. 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, KLEE can find out there is only one possible path with emitting a concrete input$(x, y) = (-1335664895, -811818755)$. As this only path passes assertion, we have proved the correctness of the code fragment.

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 code parts to check whether the hypothesis invariant $\mathcal{H}$ is an adequate inductive invariant. If not, the teacher offer an example to help the learner in the next learning round.

With these feedbacks, the learner check whether the offered example is a program counter-example. If yes, there are bugs in program which need to be fixed by developers. Otherwise the learning starts the next learning round. This will last until the hypothesis passes all the three equations or the teacher finds out a real counter-example to disprove the program.

F. Contributions and Future Work

The main contribution of this paper is to propose an new loop invariant inference framework based on active learning. It extends ICE-learning, but converge faster than the latter one. We exploit one program execution to get a state chain rather than one state or state-pair, 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 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 use SVMs to do the learning. Although we can only learn linear invariants for this moment, we can generate more complex arithmetic invariants when projecting original samples to high dimensional spaces. But there is no powerful verifier which can reasoning about complex function straightly, so there is still a lot to do before proving. We plan to use several linear inequalities to approximate the complex curve, and then our framework could be powerful enough to get most arithmetic invariants in theory.

II. An Example

This section shows 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 loop invariants, we need to prepare program ready for recording program states. We apply a simple instrumentation to the target program source code in order to get some information from program executions. In this step, we record variables' values in every loop iteration (which can be viewed as a state chain when put together) and we can also be aware of whether the first state satisfies the precondition and the last state satisfies the post-condition or not by replacing assume and assert functions (actually they are macros, not functions) with our own implementation.

```
  int x, y, xa, ya;
> assume (xa + 2 * ya >= 0);
  while (nondet()) {
+   record_sample (xa, ya);
    ... // put loop body here
  }
+ record_sample (xa, ya);
> assert (xa + 2 * ya >= 0);
```

Listing 3. Loop Example after Instrumentation

B. Active learning using SVMs

This is the main step for invariant inference in our framework. In this step, the teacher runs the target program with inputs from random number generators or membership queries submitted by the learner. Here for example, after running the instrumented target program with four randomly generated inputs $(x, y) = (9, -7), (7, 1), (-9, 12), (-15, 6)$, the teacher can provide four program state chains with their labels figured out according to Table. I. The results are listed as follows:

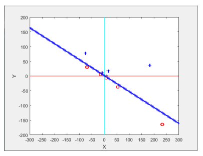
- failP (9, -7) (54, -37) (233, -164) failQ [-]
- passP (7, 1) (16, 17) (183, 36) passQ [+]
- passP (-9, 12) (-76, 78) (-217, 311) passQ [+]
- failP (-15, 6) (-70, 30) (-367, 161) failQ [-]

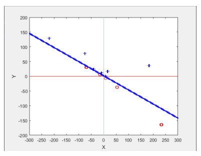
With these state chains, the learner can apply linear SVMs to learn a linear classifier to be regarded as a hypothesis invariant. The hyper-plane $\mathcal{H}_1$ (with its margin) is shown in Fig. 2(a).

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

Then the learner checks whether any implication chain can refute the hypothesis. As there is no implication chains generated by program executions so far, we can simply ignore this checking step now.

Then the learner picks up some most informative points, which lie on or near the invariant margin, to invoke membership queries. Here for instance, the learner starts a membership query with $(x, y) = (-41, 24)$.


(a) $\mathcal{H}_1$


(b) $\mathcal{H}_2$

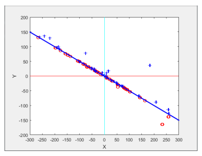

(c) $\mathcal{H}_{15}$

Fig. 2. Refining Visualization

Having membership queries from the learner, the teacher execute the target program with inputs extracted from queries' contents, and can easily get the following outputs:

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

With these new samples, the learner repeats to learn the classifier again to get a better invariant hypothesis $\mathcal{H}_2$ as shown in Fig. 2(b). As this process continues, the learner gains more and more confidence with his hypothesis.

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

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. Finally, the learner gets the hypothesis invariant $\mathcal{H}_{15}$ he needs as shown in Fig. 2(c) :

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

This learning process described above is available as animation by running the demo matlab script in the project root directory.

C. Validation by KLEE

As KLEE cannot deal with variables of double type by now, we need to round-off the coefficients in our hypothesis $\mathcal{H}_{15}$ and convert into a simpler form as follows:

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

To validate hypothesis $\mathcal{H}_c$, we divide the target program into three separated parts based on their roles in loop expression:

```
// Code Part 1: Before Loop
klee_assume (xa + 2 * ya >= 0); //precondition
klee_assert (xa + 2 * ya >= -1); //hypo-inv

// Code Part 2: Loop Body
klee_assume (xa + 2 * ya >= -1); //hypo-inv
... // put loop bady here ...
klee_assert (xa + 2 * ya >= -1); //hypo-inv

// Code Part 3: After Loop
klee_assume (xa + 2 * ya >= -1); //hypo-inv
klee_assume (xa + 2 * ya >= 0); //post-condition
```

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 concrete values as initial inputs, we realize this is actually a counter-example: it satisfies precondition, but fails at post-condition after executing the loop body once. (The exact trace is the execution chooses the else-branch, and fails at loop condition checking and thus jumps out of the loop afterwards with $(xa, ya) = (580522691, 1676896317)$, and finally fails at post-condition due to integer addition overflow exception). Hence, it becomes developers' job to locate and fix potential bugs with the help of this emitting counter-example.

In a nutshell, after the preprocessing, learning and checking phases, our framework finds out a counter-example which can trigger the assertion failure in this illustrative example. With this bug fixed, we can reapply our framework to prove program correctness or find a new counter-example indicting 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.