5-2017

Assertion generation through active learning

Long H. PHAM

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Jun SUN

Citation

PHAM, Long H.; SUN, Jun; and SUN, Jun. Assertion generation through active learning. (2017). *ICSE '17: Proceedings of the 39th IEEE/ACM International Conference on Software Engineering: Buenos Aires, Argentina, May 20-28*. 155-157.
Available at: https://ink.library.smu.edu.sg/sis_research/4706

2017 IEEE/ACM 39th IEEE International Conference on Software Engineering Companion

Assertion Generation through Active Learning

Long H. Pham, Ly Ly Tran Thi and Jun Sun
Singapore University of Technology and Design
Email: honglong_pham@mymail.sutd.edu.sg, {lyly_tranthi, sunjun}@sutd.edu.sg

***Abstract*—Program assertions are useful for many program analysis tasks. They are however often missing in practice. In this work, we develop a novel approach for generating likely assertions automatically based on active learning. Our target is complex Java programs which cannot be symbolically executed (yet). Our key idea is to generate candidate assertions based on test cases and then apply active learning techniques to iteratively improve them. The experiments show that active learning really helps to improve the generated assertions.**

I. Introduction

Assertions in programs are useful for many program analysis tasks [12]. For instance, they can be used as oracles for program testing, or correctness specification for static program verification or run-time checking. They are however often insufficiently written in practice [12], [5]. It is thus desirable to generate them automatically.

We broadly divide existing approaches for assertion generation into three categories. The approaches in the first category rely on summarizing and generalizing a set of (user-provided or generated) test cases, e.g., [7], [8], [10], [18], [11]. Typically, they are scalable and can be applied to complex programs. However, if only a limited number of test cases are available, the generated assertions are often not 'correct' [18]. The second category contains approaches which rely on some forms of symbolic execution or constraint solving, e.g., [6], [3], [2]. They often provide some guarantee on the quality of the generated assertions. However, since programs must be encoded as symbolic constraints and be solved, these approaches are often limited to relatively simple programs. The third category combines the techniques of the two categories, e.g., the guess-and-check approaches documented in [16], [15], [9] or the work in [19]. Similar to those approaches in the second category, these approaches are often limited to relatively simple programs as symbolic execution is applied.

In this work, we propose a new approach for assertion generation. Our target are complex Java programs (which often rely heavily on libraries) and thus we would like to avoid heavy-weight techniques like symbolic execution. At the same time, we would like to overcome the issue of not having sufficiently many test cases in practice and be able to generate 'correct' assertions. To do that, we use a process called active learning. Our approach has three steps as below.

II. Our Approach

Step 1: Data Collection

At each location where we want to generate assertion, we instruct the program to output the program states during the execution of the test cases. In our work, there can be two sources of test cases. The first group contains the user-provided test cases. The second group contains random test cases we generate using the Randoop approach [13].

Besides program states from executing test cases, we generate artificial program states at the location, by substituting values of variables in the current state with values generated by active learning. These states may not be reachable by any test case running from the beginning. They nonetheless may be helpful for learning assertions.

After executing the test cases with the instrumented program, we obtain a set of program states, in the form of an ordered sequence of features (a.k.a. feature vectors). We categorize the feature vectors into two sets based on the testing results, one denoted as S^+ containing those which do not lead to failure and the other denoted as S^- containing the rest.

Step 2: Classification

Intuitively, we should learn an assertion that perfectly classify S^+ from S^-. We thus borrow ideas from the machine learning community to learn the assertions through classification. We support two classification algorithms in this work. One applies the learning algorithm in [4] to learn Boolean combination of propositions generated by a set of predefined templates. The other applies Support Vector Machine (SVM) to learn assertions in the form of conjunctions of linear inequalities. Both algorithms are coupled with an active learning strategy as we discuss later.

Template based PAC Learning We adopt most of the primitive templates from DAIKON [8]. A template may contain zero or more unknown coefficients which can be precisely determined with a finite set of program states. To generate candidate assertion in the form of a primitive template, we randomly select a sufficient number of feature vectors from S^+ and/or S^- and compute the coefficients. Then we check whether the resultant predicate is valid, which means it evaluates to $true$ for all feature vectors in S^+ and evaluates to $false$ for all feature vectors in S^-.

To generate Boolean combinations of primitive templates, we start with identifying a set of predicates (in a form defined by a template) which correctly classify some feature vectors in S^+ or S^-. Then, we apply the algorithm in [4] to identify a Boolean combination of them which perfectly classifies all feature vectors in S^+ and S^-. Informally, we consider each feature vector in S^+ and S^- as data points in certain space. The algorithm works by greedily finding a set of predicates that can partition the space into regions which only contains

978-1-5386-1589-8/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE-C.2017.87

TABLE I
EXPERIMENT RESULTS ON GITHUB PROJECTS AND JAVA STANDARD LIBRARY

		ALEARNER with active learning					ALEARNER without active learning				
Project	#method	#assertion	corr	necc	suff	irre	#assertion	corr	necc	suff	irre
pedrovgs/Algorithms	96	85	69	10	2	4	88	64	15	3	6
JodaOrg/joda-time	236	133	83	43	0	7	153	25	31	37	60
JodaOrg/joda-money	93	25	16	9	0	0	30	16	9	0	5
JDK library	50	48	43	2	0	3	50	43	2	0	5

data points in S^+ or S^- but not both. Each region can be defined by a conjunction of the predicates. The disjunction of all regions containing S^+ is a perfect classifier.

SVM-based Learning In addition to template-based learning, we support learning assertions in the general form of $c_1x_1 + c_2x_2 + ... \geq k$ (a.k.a. a half space). To generate such an assertion, we need to find coefficients c_1, c_2, ... such that $c_1x_1 + c_2x_2 + ... \geq k$ for all feature vectors in S^+ and $c_1x_1+c_2x_2+... < k$ for all feature vectors in S^-. In this work, we apply SVM classification [14] to identify the coefficients.

We also can learn the conjunction of multiple half spaces by adopting the algorithm proposed in [17]. Given the feature vectors in S^+ and S^-, we first randomly select a vector s from S^- and learn a half space ϕ_1 to separate s from all vectors in S^+. We then remove all vectors s' in S^- such that ϕ_1 evaluates to $false$ given s'. Next, we select another vector from S^- and find another half space ϕ_2. We repeat this process until S^- becomes empty. The conjunction of all the half spaces $\phi_1 \wedge \phi_2 \wedge ...$ perfectly classifies S^+ from S^-.

We remark that we prefer simple assertions rather than complex ones. Thus, we first apply the primitive templates. We then apply SVM-based learning if no valid assertion is generated based on the primitive templates. Boolean combinations of templates are tried last. The order in which the templates are tried has little effect on the outcome because invalid templates are often filtered through active learning process.

Step 3: Active Learning

Once a candidate assertion is generated, we apply the idea of active learning to selectively generate feature vectors, which are then turned into program states to improve the assertion. In general, we select new feature vectors on and near the *classification boundary* of the candidate assertion. After selecting the feature vectors, we automatically mutate the program to set the program state at the location according to the selected feature vectors. Next, we run the test cases with the modified program to check whether the test cases lead to failure or not. Based on the testing results, we add new program states into S^+ or S^- and repeat the classification step to identify new candidate assertion. The process repeats until the assertion converges.

III. EXPERIMENTS

Our approach has been implemented in a tool named ALEARNER. To evaluate the effectiveness and efficiency of ALEARNER, we conduct three sets of experiments. First, we apply ALEARNER to 425 methods from three popular Java projects from GitHub. Secondly, we apply ALEARNER to a set of 50 methods in the JDK library. Lastly, we apply ALEARNER to 10 programs transformed from the software verification competition (SVComp [1]). For each method, we try to generate the assertion at the beginning of method (i.e., its precondition). The test cases for GitHub and JDK programs are user-provided, while the test cases for SVComp programs are generated randomly.

We define the correctness of an assertion in terms of whether there is a correlation between the learned assertion and whether failure occurs or not. Depending on what the correlation is, the assertions are manually categorized into four categories. An assertion is necessary if it is (only) a necessary condition for avoiding failure; it is sufficient if it is (only) a sufficient condition; and correct if it is both necessary and sufficient. Ideally, we should learn correct assertions. Lastly, an assertion is irrelevant if it is neither necessary nor sufficient.

Table 1 shows the results of ALEARNER with and without active learning for 3 projects in GitHub and JDK library. We can see that with active learning ALEARNER can learn more correct assertions and less irrelevant assertions. For SVComp programs, with active learning, ALEARNER can usually learn correct assertions for 8 programs; while without active learning, ALEARNER rarely learns a correct assertion.

On average ALEARNER takes about 40 seconds to learn an assertion, which we consider is reasonably efficient for practical usage. Without active learning, ALEARNER runs faster but only by a factor of 2, which means active learning converges relatively quickly. Given that the quality of the generated assertions improve with ALEARNER and active learning, we consider the overhead is acceptable.

There are several reasons why ALEARNER cannot learn correct assertions in some cases. One reason is the correct assertions require the templates that ALEARNER does not support currently (e.g., templates about strings with specific pattern or templates related to types of variables). Another reason is the test cases are too biased to obtain the correct assertions even with active learning. These problems can only be solved when we extend the set of templates and used more complicated techniques such as symbolic execution to generate the initial test cases.

IV. CONCLUSION

In this work, we present an approach that can infer likely assertions for complex Java programs. The novelty in our approach is to apply active learning techniques to learn and refine assertions. In the future, we would like to explore the possibility of learning full program specification with the additional help from techniques like symbolic execution.

References

[1] http://sv-comp.sosy-lab.org/2016/.

[2] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 98–109, 2005.

[3] M. Boshernitsan, R. Doong, and A. Savoia. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 169–180. ACM, 2006.

[4] N. H. Bshouty, S. A. Goldman, H. D. Mathias, S. Suri, and H. Tamaki. Noise-tolerant distribution-free learning of general geometric concepts. *Journal of the ACM (JACM)*, 45(5):863–890, 1998.

[5] P. Chalin. Rigorous development of complex fault-tolerant systems. chapter Are Practitioners Writing Contracts?, pages 100–113. Springer-Verlag, Berlin, Heidelberg, 2006.

[6] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international conference on Software engineering*, pages 281–290. ACM, 2008.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 27(2):99–123, 2001.

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[9] P. Garg, C. Loding, P. Madhusudan, and D. Neider. Ice: a robust learning framework for synthesizing invariants. 2013.

[10] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301. ACM, 2002.

[11] J. Henkel and A. Diwan. Discovering algebraic specifications from java classes. In *ECOOP 2003–Object-Oriented Programming*, pages 431–456. Springer, 2003.

[12] C. A. R. Hoare. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25(2):14–25, 2003.

[13] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *ICSE*, pages 75–84, 2007.

[14] G. Schohn and D. Cohn. Less is more: Active learning with support vector machines. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*, pages 839–846, 2000.

[15] R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *International Conference on Computer Aided Verification*, pages 88–105. Springer, 2014.

[16] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *European Symposium on Programming*, pages 574–592. Springer, 2013.

[17] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *Computer Aided Verification*, pages 71–87. Springer, 2012.

[18] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 191–200. ACM, 2011.

[19] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid. Feedback-driven dynamic invariant discovery. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 362–372. ACM, 2014.