Assertion Generation through Active Learning

Long H. Pham, Ly Ly Tran Thi, and Jun Sun

ISTD, Singapore University of Technology and Design, Singapore

Abstract. Program assertions are useful for many program analysis tasks. They are however often missing in practice. Many approaches have been developed to generate assertions automatically. Existing methods are either based on generalizing from a set of test cases (e.g., DAIKON), or based on some forms of symbolic execution. In this work, we develop a novel approach for generating likely assertions automatically based on active learning. Our targets are complex Java programs which are challenging for symbolic execution. Our key idea is to generate candidate assertions based on test cases and then apply active learning techniques to iteratively improve them. We evaluate our approach using two sets of programs, i.e., 425 methods from three popular Java projects from GitHub and 10 programs from the SVComp repository. We evaluate the 'correctness' of the assertions either by comparing them with existing assertion-like checking conditions, or by comparing them with the documentation, or by verifying them.

1 Introduction

Assertions in programs are useful for many program analysis tasks [15]. For instance, they can be used as oracles for program testing, or correctness specification for static program verification. They are however often insufficiently written in practice [15]. It is thus desirable to generate them automatically.

A variety of approaches have been developed for assertion generation. We broadly divide them into three categories. The approaches in the first category rely on summarizing and generalizing a set of test cases. One well-known example is DAIKON [11]. DAIKON takes a set of test cases as inputs and summarizes the program states at a given program location based on a set of predefined templates. Typically, these approaches are scalable and thus can be applied to complex programs. However, it is also known if only a limited number of test cases are available, the generated assertions are often not 'correct' [32]. Unfortunately, as reported in [4, 5], the number of test cases available in practice is often limited.

The second category contains approaches which rely on some forms of symbolic execution or constraint solving, e.g., [10, 6, 3]. These approaches 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 [25, 24, 13]. The idea is to *guess* candidate assertions and then *check* their correctness based on symbolic execution or similar techniques. If the candidate assertion is found to be incorrect, a counterexample is identified as a new test case and used to refine the candidate assertion. Similarly, the work in [35] generates candidate invariants and then instruments the candidate invariants into the programs. Afterwards, symbolic execution is applied to generate new test cases, which are used to improve the candidates. 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 targets are complex Java programs and thus we would like to avoid heavy-weight techniques like symbolic execution. We also would like to overcome the issue of not having sufficiently many test cases in practice and be able to generate 'correct' assertions. Briefly, our approach works as follows. We first learn some initial candidate assertions using a set of templates and machine learning algorithms (e.g., [7] and Support Vector Machine [23]). Next, we apply active learning techniques to improve the candidate assertions. That is, we automatically generate new program states based on the candidate assertions. Then, we re-learn the assertion using the testing results from new states and iteratively improve the assertions until they converge. Compared to existing approaches, our main idea is to automatically mutate program states based on active learning to refine the candidate assertions. This is motivated by recent studies in [30, 31] which show that active learning can help to learn 'correct' predicates (i.e., with bounded error) with a small number of labeled data.

Our approach has been implemented in a tool named ALEARNER. To evaluate the effectiveness and efficiency of ALEARNER, we conduct two sets of experiments. Firstly, we apply ALEARNER to 425 methods from three popular Java projects from GitHub. ALEARNER successfully generates 243 assertions. We manually inspect the generated assertions and confirm that 158 of them (65%) are correct, i.e., necessary and sufficient to avoid failure. Furthermore, we notice that 186 out of the 425 methods contain some assertion-like checking condition at the beginning of the method. For 116 of those methods (62%), ALEARNER successfully generates an assertion identical to the condition. Secondly, we apply ALEARNER to a set of 10 programs from the software verification competition (SVComp [2]). Given the postcondition in the program, we use ALEARNER to automatically learn a precondition, without any user-provided test cases. We show that for 90% of the cases, ALEARNER learns a precondition which is weaker than the user-provided precondition yet strong enough to prove the postcondition. Lastly, we evaluate the efficiency of ALEARNER and show that the computational overhead is mild.

The remainder of the paper is organized as follows. Section 2 illustrates how our approach works with examples. Section 3 presents details of each step in our approach. Section 4 presents the implementation of ALEARNER and evaluation results. Section 5 discusses the related work. Section 6 concludes.

2 Overview with Examples

In the following, we briefly describe how our approach works. Without loss of generality, we assume the input to our method is a Java method with multiple parameters (which may call other methods) as well as a set of user-provided test cases. For in-

```
public MonthDay withMonthOfYear(int monthOfYear) {
    int[] newValues = getValues();
    newValues = getChronology().monthOfYear().
    set(this,MONTH_OF_YEAR,newValues,monthOfYear);
    return new MonthDay(this, newValues);
}
...
public static void verifyValueBounds(DateTimeField field, int value,
        int lowerBound, int upperBound) {
    if ((value<lowerBound)||(value>upperBound)) {
        throw new IllegalFieldValueException(
            field.getType(), Integer.valueOf(value),
            Integer.valueOf(lowerBound), Integer.valueOf(upperBound));
    }
}
```

Fig. 1. Example from class $MonthDay$ in project $JodaOrg/joda\text{-}time$

stance, assume that the input is the method $withMonthOfYear$ shown in Figure 1, which is a method from the $joda\text{-}time$ project on GitHub. This method returns a new $MonthDay$ object based on the current object and sets its month value as the input $monthOfYear$. A series of methods are invoked through inheritance and polymorphism to create a new $MonthDay$ object, including method $verifyValueBounds$ in class $FieldUtils$ (shown in Figure 1). Method $verifyValueBounds$ checks if the value of $monthOfYear$ is within the range defined by the parameters $lowerBound$ (i.e., 1) and $upperBound$ (i.e., 12).

Our first step is *data collection*. Given a program location in the given program, we instruct the program to output the program states during the execution of the test cases. We collect two sets of program states, one containing program states which lead to failure and the other containing the rest. In the above example, assume that we are interested in generating a precondition of method $monthOfYear$, i.e., an assertion at the beginning of the method. In the project, there are three user-provided test cases for this method, with the input $monthOfYear$ being 5, 0, and 13 respectively. The latter two test cases result in failure, whereas the first runs successfully. Thus, we have two sets of program states, one containing the state of $monthOfYear$ being 0 or 13 and the other containing the state of $monthOfYear$ being 5.

The second step is *classification*. Given the two sets of program states, we apply learning techniques to identify predicates which could perfectly classify the two sets. In ALEARNER, we support two learning algorithms to identify such predicates. The first algorithm uses the predefined templates and applies the learning algorithm in [7] to learn boolean combination of the templates. The second one is inspired by [27], which applies Support Vector Machine (SVM) to learn conjunction of linear inequalities as classifiers. In our example, given the two sets of program states, applying the first algorithm, ALEARNER tries the templates one by one and identifies a candidate assertion $monthOfYear = 5$ (i.e., when $monthOfYear$ is 5, there is no failure). While this assertion is consistent with the three test cases, it is 'incorrect' and the reason is the lack of test cases. For instance, if we are provided with a test case with $monthOfYear$ being 4, assertion $monthOfYear = 5$ would not be generated. This shows that assertion

```
public Days minus(Days days) {
    if (days == null) return this;
    return minus(days.getValue());
}
public Days minus(int days) {
    return plus(FieldUtils.safeNegate(days));
}
```

Fig. 2. Example from class $Days$ in project $JodaOrg/joda\text{-}time$

generation based only on a limited set of test cases may not be effective. Using SVM, we learn the assertion: $3 \leq monthOfYear \leq 9$.

The third step is *active learning*. To solve the above problem, we apply active learning techniques to iteratively improve the assertion until it converges. This is necessary because the assertion should define the boundary between failing program states from the rest, whereas it is unlikely that the provided (or generated) test cases are right on the boundary. Active learning works by generating new states based on the current boundary. Then, we execute the program with new states to check whether they lead to failure or not (a.k.a. labeling). Step 2 and subsequently step 3 are then repeated until the assertion converges.

For simplicity, we show only how the candidate assertion $3 \leq monthOfYear \leq 9$ is refined in the following. Applying active learning, we generate two new states where $monthOfYear$ is 3 and 9 respectively. After testing, since both states run successfully, the two sets of program states are updated so that the passing set contains the states of $monthOfYear$ being 3, 5 and 9; and the failing set contains 0 and 13. Afterwards, the following assertion is identified using SVM: $2 \leq monthOfYear \leq 11$. Repeating the same process, we generate new states where $monthOfYear$ is 2 and 11 and get the assertion: $1 \leq monthOfYear \leq 12$. We then generate states where $monthOfYear$ is 1 and 12 and learn the same assertion again. This implies that we have converged and thus the assertion is output *for user inspection*.

The above example shows a simple assertion which is generated using ALEARNER. In comparison, because $withMonthOfYear$ calls many methods as well as inheritance and polymorphism in the relevant classes, applying assertion-generation methods based on symbolic execution is non-trivial[1]. ALEARNER can also learn complex assertions with disjunction and variables from different domains. An example is the precondition generated for method $minus$ in class $Days$ shown in Figure 2. Each $Days$ object has a field $iPeriod$ to represent the number of days. The method receives a $Days$ object $days$ as input. If $days$ is $null$, the method returns the $this$ object. Otherwise, it negates the number of days in $days$. Then, it returns a new $Days$ object whose number of days is the sum of the negation and the number of day in the $this$ object. An arithmetic exception is thrown when the number of days of $days$ equals $Integer.MIN\_VALUE$ or when the result of the sum is overflow. ALEARNER generates the assertion: $days = null \mid\mid (this.iPeriod - days.iPeriod\ is\ not\ overflow$ $\&\&\ days.iPeriod \neq Integer.MIN\_VALUE)$ for the method.

[1] Refer to recent development on supporting polymorphism in symbolic execution in [17].

3 Detailed Approach

In this section, we present the details of each step in our approach. Recall that the inputs include a Java program in the form of a method with multiple parameters and a set of test cases. The output is the assertions at different program locations. We assume the program is deterministic with respect to the testing results. This is necessary because our approach learns based on the testing results, which become unreliable in the presence of non-determinism.

Step 1: Data Collection Our goal is to dynamically learn likely assertions at different program locations. To choose program locations to learn, we build a control flow graph of the program and choose the locations heuristically based on the graph. For instance, we generate assertions at the beginning of a method or the end of a loop inside a method. Another candidate program location is the beginning of a loop. However, an assertion in a loop or a recursion must be inductive. Learning inductive assertions is itself a research topic (e.g., [28, 24]) and we leave it to future work.

We instruct at the program location with statements to output the program states during the execution of the test cases. In ALEARNER, there can be two sources of test cases. The first group contains the user-provided test cases. However, our experience is that often user-provided test cases are rather limited and they may or may not contain the ones which result in failure. The second group contains random test cases we generate using the Randoop approach [20]. We remark that Randoop is adopted because it is relatively easy to implement.

To collect only relevant features of the program states, we identify the relevant variables. Given a failed test case, we identify the statement where the failure occurs and find all the variables which it has a data/control dependence on through dynamic program slicing. Among these variables, the ones accessible at the program location are considered relevant. Next, we extract features from the relevant variables. For variables of primitive types (e.g., int, $float$), we use their values. For reference type variables, we can obtain values from the fields of the referenced objects, the fields of those fields, or the returned value of the inspector methods in the class. As a result, we can obtain many values from a single variable. In ALEARNER, we set the bound on the number of de-referencing to be 2 by default, i.e., we focus on the values which can be accessed through two or less de-referencing. This avoids the problem of infinite de-referencing in dealing with recursive data types.

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 then categorize the feature vectors into two sets according to the testing results, one denoted as S^+ containing those which do not lead to failure and the other denoted by S^- containing the rest. Note that the feature vectors obtained from different test cases may not always have the same dimension. For instance, in one test case, a reference type object might have the value $null$, whereas it may not be $null$ in another test case so that we can obtain more features. We then apply standard techniques to normalize feature vectors in S^+ and S^-, i.e., we mark missing features as $null$. With this normalization, all vectors have the same number of features.

Table 1. Sample Templates for assertions

Sample Template	Sample Selective Sampling
x * y = z	predefined values for x, y, and z
x = c	(x = c $\pm$ 1)
x != c	(x = c $\pm$ 1)
x = true	(x = true); (x = false)
ax + by = c	solve for x based on y and vice versa

Step 2: Classification The feature vectors in S^+ are samples of 'correct' program behaviors, whereas the ones in S^- are samples of 'incorrect' program behaviors. Intuitively, an assertion should 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 [7] to learn boolean combination of propositions generated by a set of predefined templates inspired by DAIKON. The other applies 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 Learning We first introduce our template based assertion generation approach. We adopt most of the templates from DAIKON. In the following, we first introduce the primitive templates (i.e., propositions without logical connectors) supported by ALEARNER and then explain how to learn boolean combinations of certain primitive templates.

A few sample primitive templates are shown in Table 1. In total, we have 120 primitive templates and we refer the readers to [1] for the complete list. A template may contain zero or more unknown coefficients which can be precisely determined with a finite set of program states. For instance, the template which checks whether two variables have the same value has zero coefficient and we can determine whether it is valid straightforwardly; the template which checks whether a variable has a constant value has one unknown coefficient (i.e., the constant value) which can be determined with one program state in S^+. Some templates have multiple coefficients, e.g., the template $ax + by = c$ where x and y are variables and a, b, c are constant coefficients. We need at least three pairs of x, y values in S^+ to identify the values of a, b, and c.

In order to generate candidate assertions 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. Once we compute the values for the coefficients, we check whether the resultant predicate is valid. A template with concrete values for its coefficients is called valid if it evaluates to true for all feature vectors in S^+ and evaluates to false for all feature vectors in S^-. If feature vectors are not enough to identify the coefficients, or the template requires more features than those in the feature vectors, or the template is not applicable to the input values, the template is skipped. If a template requires only a subset of the features in the feature vectors, we try all subsets of the features.

Like DAIKON, we limit the number of variables in the primitive templates to be no more than 3 and hence the number of combinations of features is cubic in the total number of features. We remark that because we learn from S^- as well, we are able to support templates which are not supported by DAIKON. One example is the template $x \neq a$ (where a is an unknown coefficient). With only program states in S^+, it is impossible to identify the value of a (since there are infinitely many possibilities). However, since the negation of $x \neq a$ must be satisfied by program states in S^-. With one feature vector from S^-, we can precisely determine the value of a.

We sometimes need assertions in the form of boolean combinations of primitive templates. In the following, we describe how to learn boolean combination of primitive templates. We start with identifying a set of predicates (in a form defined by a primitive template) which correctly classify some feature vectors in S^+ or S^-. For instance, we have the predicate $x = y$ if there is a feature vector such that $x = y$ in S^+ or $x \neq y$ in S^-. In general, it might be expensive to identify all of such predicates if the primitive template has multiple coefficients. For instance, in order to identify all such predicates in the form of $ax + by = c$, we must try all combinations of three feature vectors in S^+ to identify the value of a, b and c, which has a complexity cubic in the size of S^+. We thus limit ourselves to predicates defined by primitive templates with zero coefficient for learning boolean combination of the templates.

Once we have identified the set of predicates, we apply the algorithm in [7] 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. Each data point in S^+ is connected to every one in S^- by an edge. The problem then becomes finding a subset of the predicates (which represent lines in this space) such that every edge is cut by some predicates. The algorithm in [7] works by greedily finding the predicate which can cut the most number of uncut edges until all edges are cut. The set of predicates identified this way partition the space into regions which only contains data points in S^+ or S^- but not both. Each region is a conjunction of the predicates. The disjunction of all regions containing S^+ is a perfect classifier. We remark that DAIKON generates multiple assertions at a program location, which are logically in conjunction, and has limited support for disjunctive assertions.

SVM-based Learning In addition to template-based learning, we support learning of assertions in the general form of $c_1x_1 + c_2x_2 + \cdots \geq k$ (a.k.a. a half space) where there might be 1, 2, 3, or more variables in the expression. To generate such an assertion, we need to find coefficients $c_1, c_2, \cdots, k$ such that $c_1x_1 + c_2x_2 + \cdots \geq k$ for all feature vectors in S^+ and $c_1x_1 + c_2x_2 + \cdots < k$ for all feature vectors in S^-. With a finite set of feature vectors, we may have infinitely many coefficients $c_1, c_2, \cdots, k$ satisfying the above condition. In this work, we apply SVM classification [23] to identify the coefficients for this template.

SVM is a supervised machine learning algorithm for classification and regression analysis. We use its binary classification functionality, which works as follows. Given S^+ and S^-, it tries to find a half space $\Sigma_{i=1}^{d} c_i x_i \geq k$ such that (1) for every feature vector $[x_1, x_2, \cdots, x_d] \in S^+$ such that $\Sigma_{i=1}^{d} c_i x_i \geq k$ and (2) for every feature vector $[x_1, x_2, \cdots, x_d] \in S^-$ such that $\Sigma_{i=1}^{d} c_i x_i < k$. If S^+ and S^- are linearly separable, SVM is guaranteed to find a half space. The complexity of SVM is

$O(max(n, d) * min(n, d)^2)$, where n is the number of feature vectors and d is the number of dimensions [8], i.e., the number of values in a feature vector in S^+ or S^-.

It has been shown that SVM can be extended to learn more expressive classifiers, e.g., polynomial inequalities using the polynomial kernel and conjunctions of half spaces. In the following, we briefly describe how ALEARNER learns conjunction of multiple half spaces as the assertions (in the form of $c_1^1x_1 + c_2^1x_2 + \cdots \geq k^1 \wedge c_1^2x_1 + c_2^2x_2 + \cdots \geq k^2 \wedge \cdots$) adopting the algorithm proposed in [27]. 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 $\phi_1 \wedge \phi_2 \wedge \cdots$ perfectly classifies S^+ from S^- and is reported as a candidate assertion.

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 primitive 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, which we explain next.

Step 3: Active Learning The assertions generated as discussed above are often not correct due to the limited number of test cases we learn from, as we have illustrated in Section 2. This is a known problem in the machine learning community and one remedy for solving the problem is active learning [9].

Active learning is proposed in contrast to passive learning. A passive learner learns from a given set of data over which it has no control, whereas an active learner actively selects what data to learn from. For instance, DAIKON could be regarded as a passive learner for assertions. It has been shown that an active learner can sometimes achieve good performance using far less data than would otherwise be required by a passive learner [30, 31]. Active learning can be applied for classification or regression. In this work, we apply it for improving the candidate assertions generated by the above-discussed classification algorithms.

In the following, we explain how active learning is adopted in our work. Once a candidate assertion is generated, we selectively generate new feature vectors, which are then turned into new program states so as to improve the assertion. For template-based learning, we design heuristics to select the data on and near by the classification boundary for each template. A few examples are shown in the second column of Table 1. For example, if the assertion is $x = c$ and x is of type integer, the generated feature vectors would be $x = c + 1$ or $x = c - 1$. For templates with zero coefficients such as $x * y = z$, we choose some predefined values on and near by the boundary of $x * y = z$ as the selected feature vectors.

For SVM-based learning, we adopt the active learning strategy in [23]. The idea is to select a fixed number (e.g., 5 as in [23]) of data points on the classification boundary as the selected feature vectors. For instance, if the candidate assertion is $3x + 2y \geq 5$, we solve the equation $3x + 2y = 5$ to get a few pairs of x, y values. Note that if the candidate assertion contains multiple clauses (e.g., it is the conjunction of multiple inequalities), we apply the above strategy to each of its clauses (e.g., if it is from a template, we apply the corresponding heuristics).

After selecting the feature vectors, we automatically mutate the program so as to set the program state at the program location according to the selected feature vectors. For instance, if the selected feature vectors are $x = 4$ and $x = 6$, we generate two versions of the program. The first version inserts an additional statement $x = 4$ right before the program location in the original program, and the second version inserts the additional statement $x = 6$. Next, we run the test cases with the modified programs so as to check whether the test cases lead to failure or not. If executing a test case with the first version of the program leads to failure, the program state $x = 4$ is added to S^- or otherwise it is added to S^+. Similarly, if executing a test case with the second version leads to failure, the program state $x = 6$ is added to S^- or otherwise it is added to S^+. Afterwards, we repeat the classification step to identify new candidate assertions and then apply active learning again. The process repeats until the assertion converges.

Note that selective sampling may create unreachable states in the program. If the unreachable states are labeled negative, they do not affect the learning result because we try to exclude them. If they are labeled positive, we learn an invariant which is weaker than the 'actual' one. It is not a problem as our goal is to learn invariants which are sufficiently strong to avoid program failure.

4 Implementation and Evaluation

We have implemented the proposed method in a self-contained tool named ALEARNER, which is available at [1]. ALEARNER is written in Java with 91600 lines of code. In the following, we evaluate ALEARNER in order to answer the following research questions.

- RQ1: Can ALEARNER generate correct assertions?
- RQ2: Is active learning helpful?
- RQ3: Is ALEARNER sufficiently efficient?

As a baseline, we compare ALEARNER with DAIKON. To have a fair comparison, the experiments are set up such that ALEARNER and DAIKON always have the same set of test cases except that the test cases which result in failure are omitted for DAIKON since DAIKON learns only from the correct program executions.

Our experimental subjects include two sets of programs. The first set contains 425 methods selected from three Java projects on GitHub. Project $pedrovgs/Algorithms$ is a library of commonly used algorithms on data structures and some math operations; project $JodaOrg/joda\text{-}time$ is a library for working with date and time; and project $JodaOrg/joda\text{-}money$ is a library for working with currency. We apply ALEARNER to all classes in the first project. For the other two projects, we focus on classes in the main packages ($org.joda.time$ and $org.joda.money$) as those classes contain relatively more unit test cases. We select all methods which have at least one passed test case and one failed test case, except the constructors or the methods that are inherited without overriding (due to limitation of our current implementation). We systematically apply ALEARNER to each method, using existing test cases in the projects only. As shown in Table 5, there are a total of 2137 test cases for all the methods, i.e., on average 5 per method. *We do not generate random test cases for this set of programs*, so as to reduce randomness as well as to evaluate whether ALEARNER works with limited user-provided test cases only.

The second set contains 10 programs from the software verification competition (SVComp) repository. These programs are chosen because we can verify the correctness of the learned assertions. The programs are selected based on the following criteria. First, because ALEARNER is designed for Java programs and the programs in the repository are in C, we have to manually translate the selected programs into Java. We thus avoid programs which rely on C specific language constructs. For the same reason, we are limited to a small set of programs due to manual effort required in translating the programs. Furthermore, we skip programs with no precondition (i.e., the precondition is $true$) and non-deterministic programs. These programs are relatively small, contain relatively strong user-provided assertions (i.e., a pair of precondition and postcondition for each program) and no test cases. These 10 programs are not easy to analyze. Most of them rely on $float$ or $double$ variables and are hard to verify.

We randomly generate 20 test cases for each program. Since these programs take $float$ or $double$ type numbers as inputs which have a huge domain, we perform a simple static analysis of the postcondition, to heuristically set the range of random number generation for generating test cases. For instance, if we are to verify that some variable is always within the range of [-10, 10], we use an enlarged range (e.g., [-100, 100]) to generate input values (often for different variables). Furthermore, we manually examine the results and round the coefficients in the learned assertions to the number of decimal places that are enough to prove the postcondition based on programs specification.

All experiments are conducted in macOS on a machine with an Intel(R) Core(TM) i7, running with one 2.20GHz CPU, 6M cache and 16 GB RAM. All details of the experiments are at [1]. For all the programs, we configure ALEARNER to learn an assertion at the beginning of the method, i.e., a precondition. For each program, if random test case generation is applied, we repeat the experiment 20 times and report the average results. We set a time out of 3 minutes so that we terminate if we do not learn anything (e.g., if SVM could not find a classifier, it usually takes a long time to terminate) or active learning takes too long to converge.

RQ1: Can ALEARNER *generate correct assertions?* In this work, 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 categorized into four categories. An assertion is called *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 (i.e., there is no failure if and only if the assertion is satisfied). Ideally, we should learn correct assertions. Lastly, an assertion is called *irrelevant* if it is neither necessary nor sufficient. For instance, given a program which contains an expression `5/x`, assertion $true$ is necessary; $x \geq 2$ is sufficient; $x \neq 0$ is correct; and $x > -13$ is irrelevant.

We start with the experiment results on the GitHub projects, which are shown in Table 2. As shown in column *#asse*, a total of 243 assertions are learned by ALEARNER, i.e., ALEARNER is able to learn an assertion at the beginning of 57% of the methods. For comparison, the second last column shows the corresponding number using

Table 2. Experiment results on GitHub projects

Project	#meth	ALEARNER					DAIKON					ALEARNER wo AL				
		#asse	corr	necc	suff	irre	#asse	corr	necc	suff	irre	#asse	corr	necc	suff	irre
Algorithms	96	85	61	18	2	4	135	7	9	73	46	88	58	21	3	6
joda-time	236	133	81	49	0	3	307	7	44	71	185	153	22	42	37	52
joda-money	93	25	16	9	0	0	74	5	2	3	64	30	16	9	0	5

DAIKON. It can be observed that ALEARNER learned fewer assertions than DAIKON for all three projects. This is expected because DAIKON generates one assertion for each of its templates which is consistent with the test cases (after certain filtering [11]), whereas an assertion learned by ALEARNER must be consistent with not only the passed test cases but also the ones which trigger failure.

We first evaluate the correctness of these assertions by manually categorizing them. Table 2 shows the number of assertions in each category. Note that DAIKON often generates multiple assertions and it is often meaningless if we take the conjunction of all of them as one assertion. We thus manually check whether some assertions generated by DAIKON can be conjuncted to form correct assertions and count them as correct assertions. Necessary and sufficient assertions for DAIKON are counted similarly. Then we count the rest of DAIKON's assertions as irrelevant. In comparison, ALEARNER generates only one assertion at one program location. We can see that ALEARNER successfully generates many correct assertions, i.e., 158 out of all 243 (about 65%) are correct. In comparison, only 19 out of 516 (about 3.7%) assertions learned by DAIKON are correct, whereas majority of those learned by DAIKON are sufficient only (28%) or irrelevant (57%). This is expected as DAIKON learns based on the program states in the passed test cases only. Given that the number of test cases is limited, often the learned assertions have limited correctness.

In all three projects, ALEARNER learned more correct or necessary (i.e., over-approximation) assertions than DAIKON and much fewer sufficient or irrelevant ones. There are two main reasons why ALEARNER may not always learn the correct assertion. Firstly, ALEARNER may not always be able to perform active learning. For instance, a field of an object may be declared as $final$ and thus altering its value at runtime is infeasible. Secondly, the test cases are biased for some methods. For example, in one method in project $Algorithm$, the correct assertion should be $tree1 \neq null \mid\mid tree2 \neq null$. But in the test cases, only the value of variable $tree1$ varies (e.g., being $null$ in one test and being not $null$ in another) and variable $tree2$ remains the same. As a result, ALEARNER learns the assertion $tree1 \neq null$, which is sufficient but not necessary.

There are cases where ALEARNER cannot learn any assertion. The reason is the correct assertions require templates which are currently not supported by ALEARNER. For example, there are multiple methods which take $String$ objects as inputs and throws $RuntimeException$ if the input $String$ objects do not follow certain patterns, such as patterns for scientific numbers in $Algorithm$ and patterns for day and time format in $joda\text{-}time$. In another example, multiple methods throw $RuntimeException$ if and only if an input object is not of a type which is a subclass of certain class.

ALEARNER does not support templates related to typing and cannot learn those assertions.

We observe that, for 186 out of these 425 methods, the authors have explicitly put in code which is used to check the validity of the inputs (which is used to prevent the inputs from crashing the program by causing $RuntimeException$). This provides an alternative way of evaluating the quality of the learned assertion. That is, we assume the conditions used in these checking code are correct assertions and compare them with the learned assertions. For 116 out of the 186 (62%) methods, the assertion learned by ALEARNER is the same as the checking condition. In comparison, for only 8 out of the 186 (4.3%) methods, the condition is one of those assertions generated by DAIKON for the respective method.

Next, we evaluate the assertions generated for the SVComp programs. We formally verify the correctness of the learned precondition (by either existing program verifier or referring to the original proof of the program). Table 3 shows the experiment results. Column $correct$ shows how many times (out of 20) we learn the correct assertion. The reason that ALEARNER may not always learn the same assertion is the random test cases could be different every time. Column *useful* shows the number of times we learn a useful assertion, i.e., a sufficient condition for proving the postcondition which is implied by the given precondition. It is useful as it can be used to verify the program indirectly.

We first observe that DAIKON failed to learn any correct or useful assertion for these programs with the same passing test cases. One reason is because these programs require some precise numerical values in the assertions which are often missing from the randomly generated test cases. For 9 programs, ALEARNER learns useful assertions most of the time; and for 8 programs, ALEARNER learns the correct assertions. Further, for all these 8 cases, ALEARNER learns correct assertions which are strictly weaker than the corresponding precondition, which implies that with ALEARNER's result, we prove a stronger specification of the program. For program exp_loop, ALEARNER learned the assertion $a \neq 0$, which is implied by the given precondition $a \geq 1\mathrm{e}{-10}$ && $a \leq 1\mathrm{e}10$. However, it is necessary but not sufficient to prove the postcondition $c \geq 0$ && $c \leq 1\mathrm{e}6$. A closer look reveals that the postcondition is violated if a is greater than $2.1\mathrm{e}12$ or less than $-2.1\mathrm{e}12$. Because we never generated a random test case with such huge number, ALEARNER failed to learn the correct assertion. For program $square\_8$, we discover that the correct assertion contains two irrational number coefficients, which is beyond the capability of ALEARNER.

Based on the experiment results discussed above, we conclude that the answer to RQ1 is that ALEARNER can learn correct assertions and does so often.

RQ2: Is active learning helpful? To answer this question, we compare the performance of ALEARNER with and without active learning. The results are shown in the last columns of Table 2 and Table 3. Without active learning, the number of learned assertions and irrelevant ones increases. For instance, for methods in $joda\text{-}time$, the number of irrelevant assertions increases from 3 (i.e., 2%) to 52 (i.e., 34%). Furthermore, without active learning, we almost never learn correct assertions for the SVComp programs. This is expected as without active learning, we are limited to the provided test cases and many templates cannot be filtered. As the correct assertions for these programs contain

Table 3. Experiment results on SVComp programs

subject	ALEARNER useful	ALEARNER correct	DAIKON useful	DAIKON correct	ALEARNER wo AL useful	ALEARNER wo AL correct
exp_loop	0	0	0	0	0	0
inv_sqrt	20	20	0	0	0	0
sqrt_biN	12	11	0	0	1	0
sqrt_H_con	15	15	0	0	0	0
sqrt_H_int	13	12	0	0	0	0
sqrt_H_pse	15	13	0	0	0	0
sqrt_N_pse	13	10	0	0	0	0
square_8	15	0	0	0	0	0
zono_loose	16	16	0	0	9	1
zono_tight	14	14	0	0	11	2

Table 4. DAIKON results with selective sampling

	corr	necc	suff	irre
Without AL test cases	0	0	28	13
With AL test cases	0	0	2	8

specific numerical values, active learning works by iteratively improving the candidate assertions until the correct numerical values are identified.

Next, we conduct experiments to see whether the additional programs states generated by ALEARNER during active learning could be used to improve DAIKON. The rationale is that if it does, active learning could be helpful not only for ALEARNER but also DAIKON. We randomly selected about 10% of the methods (43 of them), created additional test cases based on the new program states, then feed those test cases (together with the provided ones) to DAIKON. The results are shown in Table 4. We can see that with additional test cases, DAIKON can filter a lot of sufficient and irrelevant assertions. We conclude that active learning is helpful for ALEARNER and may potentially be helpful for DAIKON.

RQ3: Is ALEARNER *sufficiently efficient?* To answer this question, we would like to evaluate whether the overhead of active learning is acceptable. Table 5 shows the execution time of ALEARNER (with and without active learning) as well as DAIKON's. In addition, we show the lines of the code in the projects and the number of test cases we use to analyze methods since they are relevant to the efficiency. It can be observed that ALEARNER is slower than DAIKON (about one order of magnitude), which is expected as ALEARNER relies on learning algorithms which are more time consuming than template matching in DAIKON. On average ALEARNER takes about 40 seconds to learn an assertion, which we consider as 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 active learning, we consider the overhead is acceptable.

Table 5. Experiment results on efficiency

Project	LOC	#tests	ALEARNER(w/wo AL)(s)	DAIKON(s)
Algorithms	6512	414	2496/1682	223
joda-time	85785	1163	5970/4701	665
joda-money	8464	560	1947/1739	236
SVComp	276	200	471/193	22

Threat to Validity Firstly, we acknowledge that the subjects used for evaluation might be biased. Though the three GitHub projects are selected randomly, they may not be representative of other projects. So are the programs from the SVComp repository. Secondly, although we did our best to configure DAIKON to achieve its best performance, it is not impossible that experts on DAIKON may be able to tune it for better performance. The issue of lacking test cases is a fundamental limitation for DAIKON.

5 Related Work

This work is closely related to the line of work on dynamic invariant generation, a technique pioneered by Ernst *et al.* to infer likely invariants. In particular, ALEARNER is inspired by DAIKON [11, 12]. DAIKON executes a program with a set of test cases. Then it infers precondition, postcondition, and loop invariant by checking the program states against a set of predefined templates. The templates that satisfy all these program states are likely invariants. Nguyen *et al.* extends DAIKON's approach by proposing some templates that can describe inequality, nested array [18], and disjunction [19]. They also propose to validate the inferred invariants through k-induction.

ALEARNER is different from the above-mentioned approaches. Firstly, above approaches learn invariants through summarising the program states of the passed test cases using some templates. ALEARNER learns not only from the passed test cases but also the failed ones. Therefore, it is able to learn assertions with a number of templates which cannot be supported otherwise. Secondly, ALEARNER relies on active learning to overcome the lack of user-provided test cases, which we believe is a threat to the usefulness of the above-mentioned test cases based learning tools.

Our approach is related to iDiscovery [35], which improves invariants in DAIKON by generating more test cases based on current candidate invariants and symbolic execution. In comparison, ALEARNER avoids symbolic execution. Moreover, because iDiscovery uses DAIKON to generate invariants, it only learns from passed test cases. Xie and Notkin also propose an approach similar to ours, in which test cases generation and specification inference are enhanced mutually [34]. Their work, however, does not provide any experiment results.

Sharma *et al.* proposed a number of guess-and-check approaches to infer loop invariants. They categorize program states into two sets of good and bad states. Several learning algorithms are used to learn a predicate that can separate these two sets, such as PAC learner [26], null space [25], or randomised search [24]. The predicate is then checked by a verifier to see if it is valid loop invariant. If it is not, verifier returns a counterexample and the counterexample is used to improve the learned predicate. Garg *et al.*

extend above idea by introducing ICE framework [13] and a method to learn invariants by solving a SMT formula. A new method using decision tree to learn invariants in ICE framework is presented in [14]. Krishna *et al.* also use decision tree to learn invariant in their approach [16]. These guess-and-check methods can infer correct invariants. However, they rely on the program verification and thus are limited to relatively simple programs. In comparison, our approach relies on machine learning techniques.

Padhi *et al.* present the idea of learning precondition to avoid exception with a method that can add more features in the learning process automatically [21]. In [22], the authors use decision tree to learn likely precondition from a partial truth table of a set of predicates. Lastly, our idea of using SVM to learn assertions is inspired by [27, 33, 29]. However, those works have very different goals from this one.

6 Conclusion

In this work, we present an approach that can infer likely assertions from complex Java programs. The novelty in our approach is to apply active learning techniques to learn and refine assertions. While active learning helps to overcome the issue of lacking test cases in many cases, the effectiveness of ALEARNER is still dependent on the availability of certain test cases. For instance, if a failure occurs only if some complex path conditions are satisfied and there are no test cases for triggering that exception, the condition to avoid that failure will not be learned. To solve the problem, we would like to use more systematic test case generation techniques to get better initial test cases.

Acknowledgments. This research was funded by the project T2MOE1704.

References

1. http://sav.sutd.edu.sg/alearner.
2. http://sv-comp.sosy-lab.org/2016/.
3. R. Alur, P. Černỳ, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL*, pages 98–109. ACM, 2005.
4. M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their IDEs. In *ESEC/FSE*, pages 179–190. ACM, 2015.
5. M. Beller, G. Gousios, and A. Zaidman. How (much) do developers test? In *ICSE*, pages 559–562. IEEE, 2015.
6. M. Boshernitsan, R. Doong, and A. Savoia. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In *ISSTA*, pages 169–180. ACM, 2006.
7. N. H. Bshouty, S. A. Goldman, H. D. Mathias, S. Suri, and H. Tamaki. Noise-tolerant distribution-free learning of general geometric concepts. *JACM*, 45(5):863–890, 1998.
8. O. Chapelle. Training a support vector machine in the primal. *Neural Computation*, 19(5):1155–1178, May 2007.
9. D. Cohn. Active learning. In *Encyclopedia of Machine Learning*, pages 10–14. 2010.
10. C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *ICSE*, pages 281–290. ACM, 2008.
11. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

12. 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.
13. P. Garg, C. Loding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *CAV*, pages 69–87. Springer, 2014.
14. P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In *POPL*, pages 499–512. ACM, 2016.
15. C. A. R. Hoare. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25(2):14–25, 2003.
16. S. Krishna, C. Puhrsch, and T. Wies. Learning invariants using decision trees. *arXiv preprint arXiv:1501.04725*, 2015.
17. L. Li, Y. Lu, and J. Xue. Dynamic symbolic execution for polymorphism. In *CC*, pages 120–130. ACM, 2017.
18. T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. DIG: A dynamic invariant generator for polynomial and array invariants. *ACM Transactions on Software Engineering and Methodology*, 23(4):30, 2014.
19. T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to generate disjunctive invariants. In *ICSE*, pages 608–619. ACM, 2014.
20. C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84. IEEE, 2007.
21. S. Padhi, R. Sharma, and T. Millstein. Data-driven precondition inference with learned features. In *PLDI*, pages 42–56. ACM, 2016.
22. S. Sankaranarayanan, S. Chaudhuri, F. Ivančić, and A. Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In *ISSTA*, pages 295–306. ACM, 2008.
23. G. Schohn and D. Cohn. Less is more: Active learning with support vector machines. In *ICML*, pages 839–846, 2000.
24. R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *CAV*, pages 88–105. Springer, 2014.
25. R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, pages 574–592. Springer, 2013.
26. R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *SAS*, pages 388–411. Springer, 2013.
27. R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *CAV*, pages 71–87. Springer, 2012.
28. F. Somenzi and A. R. Bradley. IC3: where monolithic and incremental meet. In *FMCAD*, pages 3–8, 2011.
29. J. Sun, H. Xiao, Y. Liu, S. Lin, and S. Qin. TLV: abstraction through testing, learning, and validation. In *ESEC/FSE*, pages 698–709. ACM, 2015.
30. S. Tong and E. Y. Chang. Support vector machine active learning for image retrieval. In *MULTIMEDIA*, pages 107–118. ACM, 2001.
31. S. Tong and D. Koller. Support vector machine active learning with applications to text classification. *Journal of Machine Learning Research*, 2:45–66, 2001.
32. Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *ICSE*, pages 191–200. ACM, 2011.
33. H. Xiao, J. Sun, Y. Liu, S. Lin, and C. Sun. Tzuyu: Learning stateful typestates. In *ASE*, pages 432–442. IEEE, 2013.
34. T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *FATES*, pages 60–69. Springer, 2003.
35. L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid. Feedback-driven dynamic invariant discovery. In *ISSTA*, pages 362–372. ACM, 2014.