11-2017

Learning likely invariants to explain why a program fails

Long H. PHAM

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

Lyly Tran THI

Jingyi WANG

Xin PENG

Citation

PHAM, Long H.; SUN, Jun; THI, Lyly Tran; WANG, Jingyi; and PENG, Xin. Learning likely invariants to explain why a program fails. (2017). *ICECCS 2017: 22nd International Conference on Engineering of Complex Computer Systems: Fukuoka, Japan, November 5-8: Proceedings*. 70-79.
Available at: https://ink.library.smu.edu.sg/sis_research/4705

Learning Likely Invariants to Explain Why a Program Fails

Long H. Pham*, Jun Sun*, Ly Ly Tran Thi*, Jingyi Wang*, Xin Peng†
*ISTD Pillar, Singapore University of Technology and Design, Singapore
†School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China

***Abstract*—Debugging is difficult. Recent studies show that automatic bug localization techniques have limited usefulness. One of the reasons is that programmers typically have to understand why the program fails before fixing it. In this work, we aim to help programmers understand a bug by automatically generating likely invariants which are violated in the failed tests. Given a program with an initial assertion and at least one test case failing the assertion, we first generate random test cases, identify potential bug locations through bug localization, and then generate program state mutation based on active learning techniques to identify a predicate "explaining" the cause of the bug. The predicate is a classifier for the passed test cases and failed test cases. Our main contribution is the application of invariant learning for bug explanation, as well as a novel approach to overcome the problem of lack of test cases in practice. We apply our method to real-world bugs and show the generated invariants are often correlated to the actual bug fixes.**

I. Introduction

Debugging is an important part of software engineering and often considered to be difficult. Software engineering is the process of constructing a program based on a specification. The specifications, which assert what is considered correct or otherwise buggy, may be missing in practice and may only exist in the programmer's mind. Ideally, if a specification that documents what is to be achieved for each statement is available, we can define a "bug" to be the first statement in the program where it fails to refine the specification. Debugging then can be done by contrasting the program against its specification to identify the first location where they differ. Without the specification, we are left with observations associated with the bug, e.g., which statements are executed in a failed test case; which statements are frequently executed in failed test cases; which conditions in the conditional statements are essential for reproducing the bug, etc. Based on these observations, extensive studies on bug localization have been conducted. Interested readers are referred to [73] for a survey of work prior to 2009 and [54], [65] for some recent attempts. However the recent studies in [59], [75] suggest that bug localization may not be sufficient as *programmers have to understand the bug before fixing it*.

Inspired by their work, we propose a method to complement existing bug localization techniques in this work. We develop a software toolkit called ZIYUAN to automatically generate likely invariants which are violated in the failed program execution. The goal is to help the programmers develop a high-level understanding of a bug. Given a program (e.g., a Java method) with an assertion and at least one test case failing the assertion, ZIYUAN first generates a set of test cases (by randomly instantiating the method parameters). Next, applying bug localization techniques [17], ZIYUAN identifies a list of ranked likely bug locations. ZIYUAN then attempts to learn likely invariants for explaining the bug at these locations one-by-one. In particular, ZIYUAN categorizes the program states at the location of all test cases into two sets, one containing the program states of those passed test cases and the other containing those of the failed test cases. Afterwards, ZIYUAN employs machine learning techniques to learn a classifier between the two sets. Intuitively, the classifier is a likely invariant which explains the difference between the the passing test cases and the failing ones.

One essential problem of this approach is the lack of test cases, i.e., we might have only a very limited set of program states at a program location. In particular, if the likely bug location is in the middle of the program, it is in general hard to generate test cases to reach the location. As a result, the learned classifier is biased and may not be useful. To solve this problem, ZIYUAN applies selective sampling [57], to iteratively generate "artificial program states" at the learning program location so as to learn a better classifier. That is, given a program location and a classifier for the program states (of the passed and failed test cases), we apply selective sampling to automatically compute the most informative program state for improving the classifier. ZIYUAN then automatically mutates the program according to the computed program states, and re-runs the test cases. Based on the testing results, ZIYUAN labels the program state accordingly (as either causing assertion failure or not) and refines the classifier. In this way, the classifier converges. We remark that these program states are artificial as they may not be reachable from the beginning of the program. Nonetheless, we show that the learned predicate correctly classifies program states at the program location and is useful in helping programmers understand the bug, as we show in the empirical studies. If we fail to find a classifier at a program location, ZIYUAN takes another potential bug location and starts the same process from there. ZIYUAN terminates when a predicate (i.e., a likely invariant) is identified or after exhausting the bug locations. The identified likely invariant is then presented to the user as a bug explanation.

To evaluate the effectiveness of ZIYUAN, we apply ZIYUAN to real-world bugs from open source projects and evaluate the generated bug explanations. Firstly, we show that the generated predicates are often correlated with the actual bug fixes. Then, we manually check whether the predicates always hold after the bug fixes or whether specific code is introduced in the fixed programs to handle the case when the predicate is not satisfied. We present detailed findings which suggest the usefulness of the generated predicates in bug comprehension. Secondly, as ZIYUAN works by learning likely invariants, we compare ZIYUAN with established invariant inference tools like Daikon [30] as well as FailureDoc [79] to show the difference. We further show, with examples, that ZIYUAN complements existing bug localization techniques [73]. Lastly, we conduct a user study by asking programmers to fix buggy programs with or without the help of ZIYUAN. The result shows that the predicates generated by ZIYUAN help bug understanding and fixing.

The rest of the paper is organized as follows. Section II presents the details of our approach using a running example. Section III presents the implementation of ZIYUAN and the results of the empirical studies. Section IV concludes with a review of related work.

II. Our Approach

In this section, we present details of our approach. We assume that the given program is deterministic, i.e., it is sequential and does not contain random number generation and there is no test harness problem. This assumption is necessary as our approach learns based

978-1-5386-2431-9/17 $31.00 © 2017 IEEE
DOI 10.1109/ICECCS.2017.12

TABLE I
TEST CASES

test	Test Input Description	Pass/Fail	Ranked Features
1	3 student objects with IDs 1,2,3 and scores 94, 60 and 100	Fail	[94,94,60,100,1,0,2,0,3,0]
2	3 student objects with IDs 3,2,1 and scores 75, 90 and 80	Pass	[90,75,90,80,3,0,2,0,1,0]
3	3 null student objects	Irrelevant	-
4	3 student objects with IDs 99,-10,0 and scores -33, 12 and 0	Pass	[12,-33,12,0,99,0,-10,0,0,0]

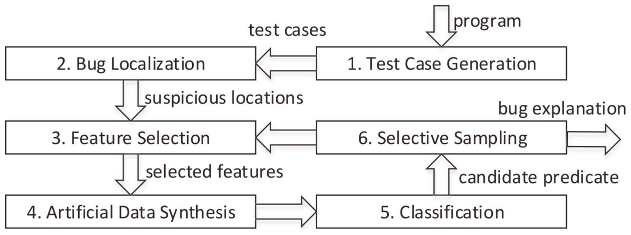

Fig. 1. Overall workflow

```
public static void program (Stu s1, Stu s2, Stu s3) {
1.  Stu[] list = new Stu[]{s1,s2,s3};
2.  standardize(list);
3.  assert(s3.newscore <= 100);
}
private static void standardize(Stu[] stus) {
4.  int max = Integer.MIN_VALUE;
5.  for (int i = 0; i < stus.length-1; i++) {
6.     if (max < stus[i].score) {
7.        max = stus[i].score;}
    }
    //version 1: max = 94;
    //version 2: max = 90;
    //version 3: max = 12;
8.  for (Stu stu: stus)
9.     stu.newscore = Math.sqrt((100-max)+stu.score)*10;
}
class Stu {
    int score; int ID; double newscore;
    public Stu (int s, int id) {score = s; ID = id;}
}
```

Fig. 2. An illustrative Java program

on testing results. The workflow of ZIYUAN is shown in Figure 1. There are 6 steps, which are explained in sequence in the following.

The program shown in Figure 2 is a toy example we designed to convey how ZIYUAN works. The program contains a method $program$ which takes multiple objects of type Stu (i.e., representing a student) as input and invokes method $standardize$ so as to standardize the students' scores through a crafted formula at line 9. Method $standardize$ takes an array of student objects and finds out the maximum score among the students and sets a new standardized score for each student in the array. We can *manually* infer that the new score is always no more than 100.

A test case of a given program is a concretization of the parameters. Let us assume that a failed test case (test 1 in Table I) is given for the above program, with the input being three student objects with scores of 94, 60 and 100 respectively. The tester notices that the third student's new score is more than 100, which signals a bug in the code. Intuitively, this is because the last student object is missed when the maximum score max is calculated, i.e., the bug manifests if the last student is the top scorer. The program can be fixed in different ways, e.g., at line 5 by changing the loop condition to $i < stus.length$, or at line 4 by setting max to be the last student's score.

In order to use ZIYUAN, first the user is asked to provide an initial assertion *in the program* based on the failed test case. For instance, we assume that the assertion at line 3 in Figure 2 is added, which asserts that the third student's new score should not be more than 100. With this assertion, the failed test case results in assertion violation. We acknowledge the difficulty in writing assertions in general [31] and remark that writing an assertion to capture the failure of a particular test case is often easier.

A. Step 1: Test Case Generation

ZIYUAN works better with a comprehensive set of test cases. In practice, the set of user-provided test cases are often limited. Thus, in order to provide more initial data for bug localization as well as classification (as explained later), ZIYUAN embeds an implementation of the Randoop algorithm [58] for random test case generation. Given a Java program, which is a method with multiple parameters, ZIYUAN generates arguments automatically for the method call. We refer the readers to the work in [58] for details. We choose Randoop over other testing techniques because it is relatively (computationally) cheap. A systematic or more sophisticated testing method (e.g., dynamic symbolic execution [37], [25] or genetic algorithm guided testing [33]) would possibly generate better test cases and improve ZIYUAN's performance.

For the running example, let us assume three test cases are generated randomly, as shown in Table I (test 2, 3, and 4). In particular, test 2 does not trigger assertion violations. In test case 3, we assume that all three student objects are null. This could be the case since test cases are generated randomly. Executing test case 3 leads to an exception but not the assertion failure and we categorize it as irrelevant (i.e., it does not reach the assertion and we have no idea whether it would have satisfied it or not). Test case 4 has three student objects with unusual IDs and scores. This is possible as we do not have a specification on the range of scores and IDs.

B. Step 2: Bug Localization

The user-provided assertion can be considered as the very first bug explanation. However, it may not be intuitively associated to the cause of the bug if it is far away from the bug, i.e., whatever misbehavior the bug has caused may have been transformed out of shape through the subsequent statements. Therefore, in this step, we identify potential bug locations in the program so that we may generate bug explanations close to where the bug is in the code.

ZIYUAN first applies a program slicer [8] to identify the statements upon which the assertion has dependencies (including both control dependency and data dependency). In our running example, this includes all numbered statements. Next, we adopt existing bug localization techniques [73] to offer clues on where the bug might be among those statements. In this work, we adopt Ochiai's approach [17], which is a spectrum based fault localization (SBFL) method. SBFL techniques are designed based on the intuitive idea: the more a statement is executed by the passed test cases, the less likely it is a bug; and the more it is executed by the failed test cases, the more likely it is. Given a set of passed test cases and failed test cases, SBFL computes a suspiciousness score for each statement in the program (based on how often it is executed by the passed/failed test cases). Different SBFL techniques use different functions to compute the suspiciousness. Applying Ochiai's approach to our example with the four test cases, line 1,2,4 and 5 have the same suspiciousness 0.5, and line 3,6,7,8 and 9 have the same suspiciousness 0.57.

Recent empirical studies [59], [75] suggest that existing bug localization techniques are not very accurate and have limited usefulness in practice. In our work, we do not assume that bug localization is precise. Rather, ZIYUAN takes the suspicious program locations as input and attempts to generate a likely invariant at those locations one-by-one, and present the bug explanation to the users.

Furthermore, recall that we view a bug explanation as an inconsistency between the program behavior and its specification; we thus favor program locations where the program behavior can be naturally specified. For instance, if a statement in a loop has a high suspiciousness, ZIYUAN would set out to look for a bug explanation after the loop because it is easier to specify the program's behavior there than in the middle of the loop[1]. Furthermore, a block of sequential statements (without branching) often has the same suspiciousness, thus ZIYUAN groups them and tries to generate only one bug explanation after the block.

In our running example, among the likely bug location (i.e., all numbered lines), ZIYUAN attempts to identify likely invariants at three program locations, i.e., right before line 8, or right after line 1, or right before line 5. Note that since line 3 is the assertion, ZIYUAN ignores it since the initial assertion is already a good bug explanation there; line 5 is a part of the first loop and thus ZIYUAN attempts to generate a bug explanation after the first loop (i.e., right before line 8); line 8 and 9 are a part of the second loop, which is followed the assertion and thus ZIYUAN ignores them.

C. Step 3: Feature Selection

After step 2, we identify a list of program locations, which are ranked based on their suspiciousness scores. Starting with the top program location in the list, we instrument the program and execute test cases so as to collect the program states (i.e., valuation of all variables) at the location in all test cases. The number of variables accessible at a program location could be huge. ZIYUAN uses the same program slicer to identify relevant ones (i.e., the variables which the assertion has a dependency on) and prunes the rest.

Next, ZIYUAN categorizes the program states into two sets: O^- containing those program states in the failed test cases and O^+ containing those in the passed test cases. Intuitively, there must be some differences between O^- and O^+ which determines whether a test case fails or not. In this work, we view a program state as a vector of features (in the form of float-type numbers) and a difference between the program states takes the form of a predicate on the features. We then apply classification techniques from the machine learning community to identify such predicates.

In the following, we first show how to obtain features from a program state. In general, there are both numerical-type (e.g., *int*, *boolean*) and categorical-type (e.g., *Stu*) variables in Java programs. We cast the value of a numerical-type variable into a feature value. To map a categorical-type object state to numerical values, we generate a *numerical value graph* from each object type [74].

Figure 3 shows a part of the numerical value graph for object $stus$ in our running example. A rounded rectangle represents a categorical type, whereas a circle associated with the type denotes a numerical value which can be extracted from the type. For readability, each edge is labeled with an abbreviated variable name and each node is labeled with the type. Notice that a categorical type is always associated with

[1]This avoids the loop invariant generation problem [18], [41].

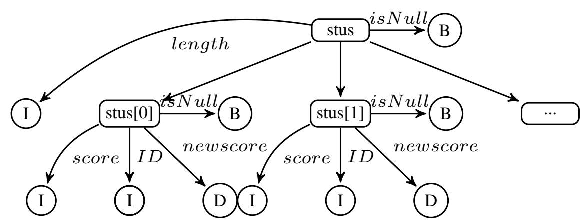

Fig. 3. The numerical value graph for object $stus$

a $boolean$ type value which is true iff the object is null. In addition, each categorical type object is associated with a set of features which are the results of the inspector methods in the respective class, e.g., the returned value of $isEmpty()$ or $length()$ for a $String$ object.

Given a program state, we can build the numerical value graph of each variable and obtain a vector of features (i.e., the numerical values in the graph) systematically. In order to apply classification techniques, each feature vector must have the same number of features. Different program states however may have different structures (e.g., two $String$ objects with different length) and therefore there are different numbers of features. In this work, we only use features which are common to program states in O^- and O^+, e.g., for arrays with different sizes, we use features like its size, the value of the first/last element, etc. The assumption is that these common features are sufficient to capture their difference. By focusing on the common features, we make sure the feature vectors are of the same size.

Another challenge is that there may be a large number of features and identifying the relevant features for generating the predicate is essential in our approach. In this work, we solve the problem heuristically by prioritizing the features based on the following two assumptions. First, we assume the recently accessed (read or written) features are more likely to be relevant. Intuitively, this is because since the bug is likely at a previous location, the features accessed recently are likely useful in explaining the bug. Thus, we sort all the features according to when they are accessed (i.e., the more recent, the higher priority). Second, we assume that the features at the top of the numerical value graphs are more likely to be relevant. Intuitively, this is because those values are easier to access and thus are more likely to be relevant to the program behavior. Thus, we further sort the features so that if two features are both not accessed recently, the one near the top of the numerical value graph has the higher priority.

Furthermore, because we prefer simple bug explanations, ZIYUAN always attempts to generate a bug explanation using fewer features, i.e., starting with one feature with top priority for classification and gradually increasing the number if necessary. That is, ZIYUAN starts by finding a classifier with the top feature; and then with the second top feature; etc., before trying to find a classifier with two or more features. For instance, in our running example, ZIYUAN tries to identify a classifier based on max's value only first; then a combination of max's value and a feature of $stus$; and so on.

We acknowledge that the features obtained this way may not always be the *best* to explain the bug. For instance, in our running example, a useful feature for explaining the bug would be the maximum score of all students, with which we can explain the bug as: the program is buggy because max is not equal to the actual maximum at line 8. Nonetheless, our empirical study shows that features obtained using the above heuristics are often be useful in explaining the bug. We plan in future work to explore alternative ways of identifying relevant features.

In our running example, given the program location right after the first loop, there are two relevant variables: max and $stus$. Since max is accessed last, it has the top priority, followed by the features of $stus$. Of all the features of $stus$, feature $score$ has higher priority since it is accessed recently in the loop. Afterwards, the top level feature on whether it is null has the higher priority than the level 1 features, and then level 2 ones. Table I column 4 shows the level 2 features of $stus$, with the value for max, for each test case.

D. Step 4: Artificial Data Synthesis

After the last step, we have transformed O^+ and O^- into two sets of feature vectors, denoted as F^+ and F^- hereafter. We then apply Support Vector Machines (SVM) to identify a predicate capturing the difference between F^+ and F^-. In order to learn an accurate classifier, a large number of samples (i.e., F^+ and F^- in our setting), are required. A limited set of samples might result in a meaningless classifier. For instance, given the data in Table I, if we use max's value to identify a classifier right before line 8, the result is: $-1 * max \geq -92$. It translates: if $max < 92$ is satisfied, there is no assertion failure. It is obviously incorrect and the reason is the lack of sufficient test cases, which is quite common in practice. In this work, we develop an approach to overcome the problem. Our approach contains two parts. One is artificial data synthesis (this step) and the other is selective sampling (step 6). In the following, we explain how artificial data synthesis works. For simplicity, we focus on learning a classifier right before line 8 in our running example.

Given F^+ and F^-, we collect all possible values of each selected feature from F^+ and F^-. Next, for each value combination of the selected features, we mutate the program by adding a statement at the program location to set the respective variables to those values. For instance, if the selected feature is max's value, based on the test cases shown in Table I, the possible values of max are 94, 90 and 12. We mutate the given program into three different versions, one by adding a line before line 8 to set max to 94; one by setting max to 90; one by setting max to 12. This is illustrated in Figure 2. Afterwards, we re-run the three test cases, for each mutated program and obtain the testing results. For instance, the additional testing results for our running example are shown in Table II, where the first row reads: setting max to 90 right before line 8 and then running test 1 results in assertion failure. Lastly, we update F^+ and F^- based on the testing results, e.g., the feature at the first row of Table II is added into F^- since the testing result is failure.

The benefit of the data synthesis is that we would have additional samples. For instance, with the additional data in Table II, $-1 * max \geq -92$ is no longer a classifier since there are both passed and failed test cases with $max = 90$. *We remark that some of the feature vectors obtained this way at the given program location are not feasible in actual execution.* For instance, there is no test case which would reach the program point with the feature vector $[12, 100, 60, 94, 1, 0, 2, 0, 3, 0]$ where $max = 12$ since 12 is not a score of any student. As a result, we would learn an over-approximation of the actual invariant (since it includes program states which are infeasible in the actual program). The additional samples are however helpful in pruning meaningless classifiers.

Figure 4 illustrates the categorization of the program states that we are getting through testing and data synthesis. It also shows the relation between the classifier that we are learning and the actual "invariant" at the program location. The circles represent the program states we obtain from the test cases (as in the running example) and the triangles represent the synthesized ones. The dashed line is to be ignored for now. There are four categories of program states: on the upper-right, we have those that lead to no assertion failure and

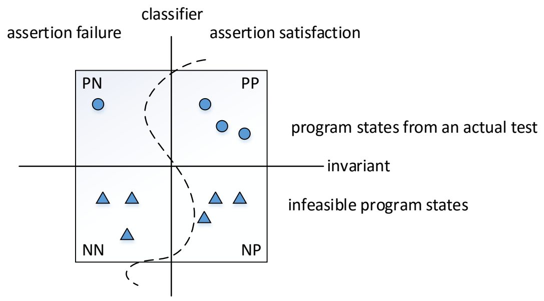

Fig. 4. Classifier vs. Invariant

TABLE II
TESTING RESULTS ON MUTATED PROGRAMS

test	mutation	Pass/Fail	ranked features
1	$max = 90$	Fail	[90,100,60,94,1,0,2,0,3,0]
1	$max = 12$	Fail	[12,100,60,94,1,0,2,0,3,0]
2	$max = 94$	Pass	[94,75,90,80,3,0,2,0,1,0]
2	$max = 12$	Fail	[12,75,90,80,3,0,2,0,1,0]
4	$max = 94$	Pass	[94,-33,12,0,99,0,-10,0,0,0]
4	$max = 90$	Pass	[90,-33,12,0,99,0,-10,0,0,0]

can be obtained from an actual test (labeled PP); in the bottom-left, we have those that lead to assertion failure and cannot be obtained from any actual test (labeled NN); and the other two (labeled PN and NP respectively). Ideally, we should rely only on program states which can be obtained from actual test cases, i.e., the upper half of the space, and we would learn $classifer \wedge invariant$. The problem is we have a limited set of test cases, in particular, we often have very few failed test cases, and as a result, the classifier would be inaccurate. By using those program states obtained from testing results on the mutated programs, we would obtain program states not only in the upper half but also the bottom half, and therefore likely a more accurate classifier, as we have witnessed in our running example. This way, the classifier we obtain would be $classifier$, which is an over-approximation of $classifer \wedge invariant$. From another point of view, the program from the program point we are investigating to the assertion is never mutated. If we take that part of the program as a function, we are feeding arbitrary inputs to that function and the classifier is a predicate on the inputs which tells whether the function would output assertion failure or not.

E. Step 5: Classification

In the following, we present how we obtain a classifier automatically based on SVM. Given two sets of feature vectors F^+ and F^-, we apply an SVM-based approach to identify a classifier between them systematically. SVM is a supervised machine learning algorithm for classification and regression analysis. We use its binary classification functionality. Mathematically, the binary classification functionality of SVM works as follows. Given F^+ and F^-, it tries to find a half space $\Sigma_{i=1}^{n} c_i * x_i \geq c$ (where c_i and c are constant coefficients and x_i are variables) such that (1) for every feature vector $[p_1, p_2, \cdots, p_n] \in F^+$ such that $\Sigma_{i=1}^{n} c_i * p_i \geq c$ and (2) for every feature vector $[m_1, m_2, \cdots, m_n] \in F^-$ such that $\Sigma_{i=1}^{n} c_i * m_i < c$. If F^+ and F^- are linearly separable, SVM is guaranteed to find a half space. Furthermore, there are usually multiple half spaces that can separate F^+ from F^-. In this work, we always choose the *optimal margin classifier* (see the definition in [68]) if possible. This half space could be seen as the strongest witness why F^+ and F^- are different. If, however, F^+ and F^- cannot be perfectly classified by one half space only, we need to identify multiple half spaces, which together classifies F^+ and F^-. We adopt the algorithm in [68] to find those multiple half spaces. We remark that ZIYUAN is extensible so that different classification algorithms can be adopted.

In our running example, if we select max and whether $stus$ is null or not as the relevant features, with the data in Table I and II, the adopted algorithm finds no classifier, since $[90, 0]$ is both in F^+ and F^-. Similarly, we would find no classifier if we use only $stus$' level-1 features or only one feature from all those level-2 features, e.g., only $stus[0].ID$, or only $stus[0].score$, etc. However, if we use two features: max and $stus[0].ID$, we obtain the classifier: $0.026 * max + 0.980 * stus[0].ID \geq 4.305$ where the numbers are rounded off to three decimal places for simplicity. The classifier is consistent with respect to all the data we have in Table I and II. It is however not meaningful. In the following, we discuss how to fix this problem.

F. Step 6: Selective Sampling

The above example shows that, even with artificial data synthesis, the classifier might still be incorrect, due to the lack of samples. In fact, without feature vectors right by the 'actual' classifier, it is very unlikely that we would find the actual classifier. This is illustrated at a high level in Figure 4. There could be many classifiers separating those samples in PP and NP from those in PN and NN. The dashed line is one example of them. To get the 'actual' classifier, we need samples which would distinguish the actual one from any nearby one. This problem has been addressed in the machine learning community through active learning and selective sampling [67]. The idea is to repeatedly generate samples nearby the current classifier and then re-classify to identify an improved classifier. In particular, SVM selective sampling techniques have been shown to identify accurate classifiers in many applications [71], [72]. In the following, we present how selective sampling is applied in our work. The idea is a simplified version of [61].

Algorithm 1 presents details on how selective sampling is adopted in ZIYUAN. At line 1, we obtain a classifier by calling the adopted algorithm svm, which is in the form of a conjunction of multiple half spaces. We then apply selective sampling to compute feature vectors which are close to the classification boundary. In particular, at line 4, we apply standard techniques [67] to identify two points on the boundary of each half space. For each computed point (i.e., a feature vector), right before the program location, we mutate the program state according to the feature vector. Next, we execute the test cases and update F^- and F^+ accordingly at line 8 and 10, based on the testing results. We then call svm again to get a new classifier at line 11. If the newly identified classifier differs from the old one, we repeat the process; otherwise we return the newly identified classifier.

As presented above, in our running example, due to the very limited set of test cases, the first classifier using max's value and $stus[0].ID$ is $0.026 * max + 0.980 * stus[0].ID \geq 4.305$. We then obtain the samples: $[90, 2]$ and $[128, 1]$ by taking existing feature values and solve for the other based on the current classifier. That is, we take max to be 90 and solve $0.026 * max + 0.980 * stus[0].ID = 4.305$ and get $stus[0].ID = 2$. Similarly, we get the other pair by taking $stus[0].ID$ to be 1. Next, we mutate the program by inserting $max = 90$ and $stus[0].ID = 2$ right before line 8 in the program. We re-run the three test cases and we obtain the additional samples in Table III. Next, invoking svm returns $null$ since $[90, 2]$ is both labeled in F^+ and F^- (i.e., the same feature vectors are both positive and negative). The algorithm then returns $null$ at line 3 in the next iteration.

TABLE III
TESTING RESULTS ON SELECTIVE SAMPLES

test	mutation	Pass/Fail	L2 features
1	[90, 2]	Fail	[90,2,100,0,2,60,0,3,94,0]
1	[128, 1]	Pass	[128,1,100,0,2,60,0,3,94,0]
2	[90, 2]	Pass	[90,2,75,0,2,90,0,1,80,0]
2	[128, 1]	Pass	[128,1,75,0,2,90,0,1,80,0]
3	[90, 2]	Pass	[90,2,-33,0,-10,12,0,0,0,0]
3	[128, 1]	Pass	[128,1,-33,0,-10,12,0,0,0,0]

```
Algorithm 1: Algorithm $classify(F^+, F^-)$
   Input: $F^+$ and $F^-$
   Output: a classifier for $F^+$ and $F^-$
1  let $clf$ = $svm(F^+, F^-)$;
2  while true do
3      return $null$ if $clf$ is $null$;
4      compute the next sample $sam$ using selective sampling;
5      mutate the program according to $sam$;
6      for each test case going through the location do
7          if test fails the assertion then
8              extract $f^-$ and add $f^-$ into $F^-$;
9          else
10             extract $f^+$ and add $f^+$ into $F^+$;
11     $newclf$ = $svm(F^+, F^-)$;
12     if $newclf$ differs from $clf$ then
13         $clf$ = $newclf$;
14     else
15         return $newclf$;
```

Next, ZIYUAN tries to learn classifier with other features. For the same reason, ZIYUAN finds that there is no classifier using features like max and $stus[0].score$ (or $stus[1].score$). However, if we use max and $stus[2].score$ as the relevant features, with only the data in Table I, II and III, we obtain: $0.053{*}max{-}0.125{*}stus[2].score >= -6.058$. Next, we apply selective sampling and keep computing new samples. For instance, one new sample is [74,80]. After testing, it is added into F^-. With new labeled samples, we obtain a better divider. After multiple iterations, the algorithm terminates and reports the classifier: $2 * max - 2 * stus[2].score \geq -1$. Since both variables are integers, it is simplified as $max \geq stus[2].score$.

Intuitively, what we learned is: assertion failure occurs if $max \geq stus[2].score$ is not satisfied. To make sure the assertion is always satisfied, the programmer should examine the predicate and decide whether it should be an invariant at the location. If it is, the program before the program location should be modified such that the predicate is always satisfied. For our running example, $max \geq stus[2].score$ should be an invariant and in this case it correctly suggests that max is computed wrongly and therefore the program before line 8 must be modified. If the programmer decides that the predicate is not supposed to be an invariant, the program after the program location needs to be modified such that when the predicate is not satisfied, the assertion could still be satisfied. That is, the (negation of the) predicate captures a generalized case which is either not handled at all or not handled correctly in the program.

G. The Overall Algorithm

We present the overall approach of ZIYUAN in Algorithm 2. ZIYUAN has four configurable parameters. M is the number of ran-

Algorithm 2: The Overall Algorithm: $explain(Prog, F, P)$

```
   Input: program $Prog$; failed test set $F$ and passed test set $P$
   Output: a likely invariant
1  generate $M$ random test cases;
2  execute them and add them to $F$ and $P$ accordingly;
3  identify a list of $X$ potential bug locations;
4  for each bug location b in the list do
5      extract a set of $N$-dimension feature vectors $F^+$ from $P$;
6      extract a set of $N$-dimension feature vectors $F^-$ from $F$;
7      while there is a new combination do
8          select a combination of $K$ or less out of $N$ features;
9          apply artificial data synthesis and update $F^-$ and $F^+$;
10         let $exp = classify(F^+, F^-)$;
11         if $exp$ is not null and contains 3 clauses or less then
12             return the classifier;
13 output "no explanation is identified";
```

dom test cases to be generated; X is a threshold on the suspiciousness score, i.e., only those program locations with a suspiciousness more than X are examined; N is the maximum size of the feature vectors; and K is the maximum number of features used in a classifier. We start with generating M random test cases and categorize them into failed ones and passed ones. Next, we apply bug localization to identify a list of program locations to generate likely invariants. For each program location with suspiciousness more than X, we identity two set of ordered feature vectors. For each combination of K or less features out of a total of N features, we apply artificial data synthesis and classification and selective sampling, to search for a classifier. Anytime a classifier is identified, we terminate and report it as the bug explanation. Note that it may find a classifier composed of many half spaces, which could be complicated for user comprehension. Thus, we throw away the classifier if it contains more than a threshold number of (3 by default) half spaces. The algorithm terminates when we exhaust the program locations and features.

The classifier identified by the algorithm is always correct with respects to the feature vectors (which are either obtained through the test cases or synthesized in the process). Since there are only finitely many combinations of program locations and features, Algorithm 2 is always terminating. We roughly measure the complexity of the algorithm in term of the number of calls of the SVM classification algorithm. It is bounded by $\#X * C^K_{N+K-1}$ where $\#X$ is the number of program locations with suspiciousness more than X and C^K_{N+K-1} is an upper bound for $C^1_N + C^2_N + \cdots + C^K_N$. In practice, $\#X$ is often limited to be a small number like 10 (i.e., we examine the top 10 bug locations (after grouping consecutive ones) and K is 3 by default and N is 10 by default. As a result, the above complexity is often manageable in our experiments.

III. Implementation and Evaluation

Our approach has been implemented as a toolkit named ZIYUAN (available at [3]). ZIYUAN is built upon a number of open source software projects, including Randoop [2], Javaslicer [8], JaCoCo [16], LIBSVM [15], and Java ILP [1]. In the following, we evaluate ZIYUAN in order to answer three research questions (RQ).

Our test subjects include 21 real-world bugs from open source projects including the JavaParser1.5 project (JP), the Java-diff-utils project (JDU), the Joda-Time project (JT) and Apache Commons Math library (ACM)), from the bug collection in [46] (D4J) and the bugs discovered in [79]. These bugs are selected based on the following criteria. First, we select bugs which are relatively easy to understand. This is because we aim to manually specify the initial assertion as well as to check whether the generated predicate is relevant. Second, we select those buggy programs with at least one passed test case. Lastly, we are limited to buggy programs which do not rely on Java features which are not yet supported in ZIYUAN (e.g., abstract methods). The bugs are summarized in Table IV, where the first column shows the project name, the second column shows the issue number and the third column is the link to the bug report. Note that a '-' in the table means the information is skipped as it is irrelevant or not available.

For each bug, we manually created an initial assertion according to the bug report. This is often straightforward if the bug results in an exception, i.e., we find the line where the exception is thrown and add an assertion to turn the exception into assertion failure. For the sake of repeatable experiments, we disable random test generation for all the experiments (i.e., set M to be 0) and use only existing test cases in the projects with an additional failed test case created according to the bug report. Notice that we manually remove the assertions in the test cases so that a test case fails if and only if the assertion in the program is violated. Furthermore, we set ZIYUAN to focus on program locations with a suspicious score of 0.5 or above. ZIYUAN is set to search for a classifier constituted by at most 3 features from the top 10 features. Lastly, SVM often takes a long time if there is no linear classifier and therefore we set a 5 second time out for each invocation of SVM. Details of the projects and the bugs, along with our analysis logs can be found at [3].

RQ1: Is ZIYUAN sufficiently efficient? We first evaluate whether ZIYUAN is sufficiently efficient for practical usage. The fifth column of Table IV shows the average execution time of ZIYUAN over 10 executions for each bug. The experiments were conducted in Windows 7 on a machine with an Intel(R) Core(TM) i5-2430m, running with one 2.40GHz CPU, 4M cache and 8 GB RAM. The data shows that ZIYUAN takes a few minutes to generate the predicates, which we believe is reasonably efficient, since it usually takes hours to fix a bug [49]. To show that these bugs are not trivial (e.g., it is hard to trace the failed test case step-by-step to locate the bug), the 4th column shows the number of statements executed in the failed test case (excluding external library calls).

We remark that sound optimization have been implemented in ZIYUAN to improve its efficiency. For instance, Algorithm 1 may take many iterations to converge. In order to reduce the number of iterations, each time a classifier is identified, we make use of the type information for better selective sampling. For instance, after calculating a new sample $[x, y]$ with two integer-type features at line 4 of Algorithm 1, we additionally check and label nearby samples, for instance $[x + 1, y]$, $[x, y + 1]$, $[x - 1, y]$, $[x, y - 1]$, so that Algorithm 1 converges fast.

RQ2: Does ZIYUAN generate useful bug explanations? We acknowledge that it is subjective on whether a predicate learned by ZIYUAN is useful in explaining the bug. In the following, we attempt to answer this question in three ways. First, we check whether the predicate is relevant by manually examining the corresponding bug fixes. Second, we present specific findings for some of the bugs and the reason why we believe the bug explanation is useful, so that the readers can judge by themselves. Third, we conduct a user study to see whether the bug explanations are useful for bug understanding and fixing. We present the details below.

TABLE IV
EFFICIENCY EVALUATION

Project	Issue #	URL	LOCfail	Time	Relevance	Daikon	Ochiai vs. ZIYUAN
JP	46	[6]	707	3m	Missing Case	to	29/**3**
JP	57	[7]	1154	15m	Invariant	to	48/**39**
JDU	10	[5]	85	73s	Invariant	×	81/**6**
JT	227	[10]	1109	4m	Incorrectly Handled Case	error	**3**/55
JT	21	[9]	1113	24s	Incorrectly Handled Case	error	43/**2**
JT	77	[11]	1210	61s	Missing Case	error	54/**15**
ACM	835	[14]	18	7m	Invariant	×	**2**/3
ACM	1196	[13]	152	42s	Incorrectly Handled Case	error	152/**1**
ACM	1005	[12]	19	4m	Invariant	error	4/**1**
D4J Time	8	[46]	5	69s	Incorrectly Handled Case	error	2/**1**
D4J Math	1,4,38,40,58,61,70,79,84	[46]	-	13m(total)	Inconclusive	-	-
FailureDoc 1	-	[4]	576	33s	Incorrectly Handled Case	+	-
FailureDoc 2	-	[4]	64	75s	Missing Case	+	-

Relevance Recall that a predicate generated by ZIYUAN could be either an actual invariant (which is violated due to a bug) or a predicate that captures a generalized case which is not handled at all (i.e., a missing case) or handled incorrectly. Thus, if the generated predicate is 'correct', either the bug should be fixed such that the predicate becomes an invariant or specific code is introduced to handle the case when the predicate is not satisfied. We manually examine the bug fixes to check whether it is the case for each bug. If the answer is yes, we consider that the predicate is relevant. Notice that some of the bugs were open and thus we proposed the fixes based on our analysis and confirmed them with the authors.

The results are summarized in Table IV column "Relevance". For all bugs, the predicate generated by ZIYUAN is satisfied in all the passed test cases and is not satisfied in the failed test case. Note that for 9 bugs in ACM, due to our limited understanding of ACM's implementation, we are not yet to be able to confirm whether the generated predicate is related to the actual cause of the bug. For the rest, in 4 cases, the fixes precisely make the learned predicate an invariant at the program location. In 3 cases, the program is fixed by introducing code to handle the case when the learned predicate is not satisfied. In 5 cases, the program is modified so that it handles the case when the learned predicate is not satisfied differently. We conclude that the predicates are relevant in these 12 cases.

Specific Findings Next, we present sample findings of the bugs and the generated predicates.

The JP project aims to build a Java 1.5 parser with AST generation and visitor support. The AST records the source code structure, javadoc and comments; and supports changing the AST nodes or creating new ones. ZIYUAN is applied to analyze an open bug (issue 46) and a closed bug (issue 57) for this project.

The bug report for issue 46 contains the following information. After parsing the Java program shown below, the output of the method *CompilationUnit.toString()* in JavaParser1.5 prints only comment 3, whereas it should print all three comments.

```
/** Comment 1*/
/** Comment 2*/
/** Comment 3*/
package net.perfectbug.test;
public class Test {}
```

With the information, we first manually created a test case according to the report. Next, we added an assertion in JavaParser1.5 to assert that after parsing the above program, invoking *CompilationUnit.getComments().size()* would return more than 1 (i.e., there should be more than 1 line of comments). We then fed the program,

```
57. private void CommonTokenAction(Token token) {
58.   lastjavadoc = null;
59.   if (token.specialToken != null) {
60.      if (comments == null) {
61.         comments = new LinkedList<Comment>();
62.      }
63.      Token special = token.specialToken;
64.      if (special.kind = JAVA_DOC_COMMENT) {
65.         lastJavaDoc = ...;
66.         comments.add(lastJavaDoc);
67.      } else if (special.kind==SINGLE_LINE_COMMENT) {
68.         LineComment comment = ...;
69.         comments.add(comment);
70.      } else if (special.kind==MULTI_LINE_COMMENT) {
71.         BlockComment comment = ...;
72.         comments.add(comment);
73.      }
74.   }
75. }
```

Fig. 5. Sample code from JavaParser1.5

the failed test case, along with existing passed test cases to ZIYUAN. After program slicing, testing and learning, tracking through 7 classes, ZIYUAN outputs a message which says that the assertion is satisfied if $special.specialToken.isNull$ is true at line 67 of class $japa.parser.ASTParserTokenManager$; otherwise, it fails.

Without knowing how JavaParser1.5 is implemented, we examine the code around line 67, as shown in Figure 5. By checking the value of $special.specialToken.isNull$ in the test cases, we realize it is not true only if there are multiple consecutive comments before a token (which could be a class or statement). Furthermore, variable $comments$ contains only the last comment (not all comments) when $special.specialToken.isNull$ is not true, which according to ZIYUAN, is when a test fails. Since $special.specialToken.isNull$ being true is not likely an invariant at this program location, we conclude that it signals a missing case, i.e., the authors forgot to handle the case when there are multiple consecutive comments. We then fixed the bug by introducing a while loop to add the multiple comments one-by-one if $special.specialToken.isNull$ is not true, replacing the block from line 59 to 74 in Figure 5. The bug is then confirmed fixed (by the authors).

We also applied ZIYUAN to issue 57 which reports that a particular method signature is parsed incorrectly. Without any knowledge on how the parsing works, we added a trivial assertion (without any generalization) to say that if the input is this particular method signature, the result should be certain particular string. ZIYUAN identified a likely invariant: $type.typeArgs.isNull == true$, at line 1755 in class $ASTParser$, which reads: if $type.typeArgs.isNull$ is true, the failure does not occur. The actual fix (by the project authors) is at line 1810 (which is 4 statements before executing line 1755) and the fix is the insertion of the statement: $type.typeArgs = null$, which makes the learned predicate an invariant.

The two examples so far resulted in predicates constituted by boolean variables only. In the following, we show examples where selective sampling helps us to generate the exact boundary conditions. We applied ZIYUAN to three issues in ACM: 835, 1196 and 1005. In particular, issue 1196 is a bug which is still open. It states that if variable x is set to be *0x1.fffffffffffffp-2* (equivalent to value 0.49999999999999994), $FastMath.round(x)$ returns 1 instead of 0 while clearly $x < 0.5$. We instrumented the program to assert that if a number is less than 0.5, the rounding result should be less than 1. ZIYUAN tracked to the statement $x + 0.5$ in the program and started finding classifiers. In our first attempt, ZIYUAN failed to identify any classifier after a while. Our investigation shows that after a few iterations, the classifier becomes $x \leq 0.49991269898708784$, LIBSVM fails to classify the samples because the samples are too close. We then implemented a simple classification algorithm (and a simple solver for the same reason) to learn classifiers in the form of $x \geq c$ and obtained a predicate $x \leq 0.49999999999999991$. It means that when x is smaller than the number, the rounding result is correct. This result in fact generalizes an open bug in JDK 6 and 7 (bug number JDK-6430675) by giving a range of x which could trigger the bug. For issue 835, ZIYUAN discovered that a likely invariant $fraction.numerator >= 0$ is violated in the failed test cases, which turned out to be the result of an integer overflow. A similar discover has been made for issue 1005.

We applied ZIYUAN to analyze three issues of JT: 21, 27 and 227. Issue 227 reports that adding 50 days from May 15 results in June 4, which is clearly wrong. We added an assertion before method *AddDays* in class *MonthDay* and ZIYUAN generated the predicate $days + iValues[1] \leq 62$, which reads that if the number of days to be added plus the original day is larger than 62, the bug occurs. It points to a bug which is activated only if the resultant date is in the next-next month or later. Due to the space limit, we skip the details on ZIYUAN's findings for other bugs in the JT project or the JDU project. Interested readers are referred to [3] for the details. Though limited in the number of test subjects, we confirm ZIYUAN to be useful in helping users to understand these bugs.

User study Finally, we perform a user study to evaluate whether independent programmers consider the generated predicates useful. The user study is conducted with 12 programmers (including PhD students, research assistants and research fellows). The programmers have a various number of years of programming experience (from 2 to 9 with an average of 5.75). They were divided into two groups randomly. The programmers in the first group were instructed to fix JP issue 46 without ZIYUAN's help and then to fix JDU issue 10 with ZIYUAN's help. The other group were instructed to fix the former issue with ZIYUAN's help and then the latter issue without ZIYUAN's help. This experiment is thus similar to a scenario where ZIYUAN is used to help a programmer to fix a bug in the legacy code. These two bugs are representative and not easy to fix.

Each programmer was given at most 30 minutes to study the bug so as to figure out the reason of the bug and propose a fix. We then evaluated whether their explanation and proposal were correct. For the first bug, with ZIYUAN's help, 3 out of 6 programmers figured out the bug correctly in 10, 27, 30 minutes respectively. Without Ziyuan's help, 2 out of 6 did it in 14 and 30 minutes respectively. For the second bug, with ZIYUAN's help, 4 out of 6 programmers did it in 15, 23, 24 and 30 minutes respectively. Without Ziyuan, none of the programmers did it. Furthermore, all of the programmers agree that the information provided by ZIYUAN was helpful. We take this as a positive feedback on the usefulness of the generated predicates. We acknowledge that the user study is limited in the number of programmers and bugs. We refer the readers to [3] for the details on the user study.

RQ3: Does ZIYUAN complement existing approaches? ZIYUAN can be categorized as an invariant learning tool. Thus, we performed experiments to compare ZIYUAN with the popular invariant generator DAIKON as well as FailureDoc reported in [79]. To compare with DAIKON, we use the same set of passed test cases used in ZIYUAN for each project and check whether DAIKON can learn an invariant which is relevant (as defined above). Note that DAIKON does not learn from failed test cases. Furthermore, the 'artificial' program states generated by ZIYUAN do not constitute actual test cases and thus cannot be used by DAIKON or FailureDoc. The results are summarized in column DAIKON of Table IV, where $error$ means an exception; to means timeout after one hour; $\times$ means none of the learned invariants are relevant and $+$ means some invariants are relevant. DAIKON failed to learn useful invariants in most of the cases.

Similar to ZIYUAN, FailureDoc aims to explain a failed test case. However, it focuses on the failed test case only (without analyzing the source code) and generates a predicate constituted by variables used in the failed test case only. In a way, it can be considered as applying ZIYUAN with the following restrictions: (1) learning based on the variables in the failed test case only, using DAIKON to generate a likely invariant, and not applying selective sampling. We tried FailureDoc on the list of bugs ZIYUAN analyzed and had no useful results because FailureDoc does not support user-provided assertions. As shown above, we managed to apply ZIYUAN to some of the bugs analyzed by FailureDoc in [79] and generated useful bug explanation in the program. We conclude that FailureDoc and ZIYUAN are useful in different settings.

ZIYUAN has a different goal from SBFL. However, we show that ZIYUAN could potentially be used to improved SBFL. The last column of Table IV shows two numbers. The first one is how many statements must the user examine before reaching the statement containing the bug, assuming that the user examines the program statement-by-statement based on the suspiciousness ranking generated by Ochiai's approach. The second one is the number of statements the user has to examine, assuming the user starts with where ZIYUAN generates the bug explanation and works towards the bug following the statements executed in the failed test case. A smaller number (highlighted in bold) is better since fewer statements are to be examined. We do not have the fixes for the bugs presented in the last three rows and thus we skip them for this comparison.

First, it can be observed from the data that SBFL may not always be effective, which is consistent with the observations in [59], [73]. Second, though ZIYUAN relies on bug localization, we observed in 8 out of 10 cases that the predicate is not generated at the most suspicious program location, but a program location closer to where the bug is in the code. Intuitively, this could be explained as follows: where the bug is easier to explain may also be where the fix is easier to fix. In the case of JT issue 227, the bug explanation is far from the bug because a large part of the relevant codes are a recursive method (i.e., method add in class $BaseDateTimeField$) and ZIYUAN currently tries to explain the bug only before or after loops or recursive methods. Though the number of bugs we studied is limited, the results suggest ZIYUAN may complement SBFL.

Limitations ZIYUAN has a number of limitations. First, though artificial data synthesis and selective sampling help to overcome the lack of test cases, the quality of the generated predicate may still depend on the test cases. For instance, in the extreme case, if no other test cases other than a failed test case is provided, neither artificial data synthesis nor selective sampling would help. To overcome this limitation, we are currently working on integrating ZIYUAN with sophisticated testing engines to boost its performance.

Second, the effectiveness of ZIYUAN relies on the user-provided assertion. In general, the stronger the initial assertion is, the stronger a bug explanation might be generated. In our running example, if the assertion at line 3 is $s1.newscore \leq 100 \wedge s2.newscore \leq 100 \wedge s3.newscore \leq 100$, the learned predicate is $stus[0].score \leq max \wedge stus[1].score \leq max \wedge stus[2].score \leq max$. We are investigating how to automatically generate the initial assertion.

Third, in general we cannot guarantee that the learned predicate is satisfied *if and only if* the given assertion is satisfied. This problem can be solved by applying program verification techniques, i.e., to verify that the learned predicate is the weakest precondition of the program from the learning program location to the assertion, with respect to the assertion. Nonetheless, existing program verification techniques often have their own limitations and may not scale to complicated programs that we would like to handle.

Fourth, the effectiveness of ZIYUAN depends on identifying the right features. Although our heuristics for feature selection worked in our empirical study, in general feature selection is challenging. We are investigating whether we can use advanced program analysis or feature selection methods to identify the relevant features automatically. Furthermore, ZIYUAN currently does not use inspector method results other than those returning boolean values as features for learning. This is because, unlike instance variables which we can change their values during selective sampling, changing the returned values of inspector methods are challenging in general.

Fifth, the classification algorithm used in ZIYUAN is limited to predicates in certain form. They may not be sufficient sometimes, e.g., the actual predicate could be non-linear or disjunctive. We are currently investigating different classification algorithms (e.g., SVM with kernel methods and neutral network) to overcome this problem. The challenge however is ensuring that the learned classifier is comprehensible by programmers.

Lastly, our empirical study is limited in the number of studied subjects and varieties. We are currently extending our collections of programs and bugs for further study.

IV. Conclusion and Related Work

The main contribution of ZIYUAN is the application of invariant learning for bug explanation, as well as a novel approach to overcome the problem of lack of test cases in practice. In essence, what ZIYUAN does is to propagate the initial user-provided assertion through the program to a location that is close to where the bug is. We believe that this is useful as programmers could then compare our bug explanation with their understanding of the program specification.

This work is also inspired by the line of work by Zeller and his collaborators, e.g., [76], [77], [26], [65], [35]. In particular, this work is closely related to the work in [65]. In [65], the authors proposed to isolate bug causes through directing test case generation (based on [33]) towards certain factors which are potentially associated with the bug cause. Two kinds of factors are considered: the executed branches and state predicates. Similarly, ZIYUAN identifies the bug causes in the form of state predicates. The state predicates used in [65] (based on their previous work [34]) include comparison between accessible variable values at certain program locations, whereas ZIYUAN relies on SVM to learn more complicated predicates. This work is related to previous work on using likely invariants for debugging [66], [40], [62]. Furthermore, this work is related to partial specification generation using symbolic methods [63], [44], [45]. ZIYUAN complements the above work by using SVM to discover relevant state predicates and, novelly, a way of “testing” and refining the predicates (e.g., by selective sampling).

This work is inspired by the line of work on invariant learning by Ernest and his collaborators [30], [56], [55], [60], [79]. In particular, this work is closely related to the work documented in [79], which shares the same goal of explaining failed tests by inferring likely invariants. Their approach is to generate mutated tests based on the failed test case, obtain a set of failure-correcting objects and use DAIKON to summarize properties of the failure-correcting objects, and lastly translate them into explanatory code comments. ZIYUAN complements their work by analyzing not only the failed test case but also the code, and in the way how mutated tests are generated (e.g., selective sampling) and how the properties of the failure-correcting objects are generated.

This work is related to the work in [69], where the authors learn a model in the form of finite state-automata to represent the scenarios in which errors occur. Our work has a different goal and a different learning approach. This work is related to work on explaining counterexamples, e.g., [20] using the notion of causality and [38] which is similar to delta debugging [76], and [51]. In contrast, we focus on learning a local invariant which helps bug understanding.

This work benefited from ideas from existing work on specification learning, including [74], [70], [36], [19], [22], [36], [42], [21], [29], [43], [28]. ZIYUAN uses SVM-based learning to discover new predicates, which is similar to previous work in [74], [70]. In [74], random testing and SVM are used to learn a typestate for Java classes. Later, the work in [70] extends [74] to provide correctness and accuracy guarantee of the learned typestate. This work is different as we have a different objective (i.e., bug explanation) and a different learning approach, i.e., instead of L* [74], [70], we use active learning and selective sampling for discovering invariants. This work is related to work on inferring documentation from programs as ZIYUAN also learns program invariants. Examples include [64] which facilitates programmers to write documentations, [23] which infers documentations from exceptions, [24] from software changes, etc. Our work is different as it is motivated for bug explanation.

In addition, this work is related to research on bug/fault localization, including but not limited to [73], [54], [65], [32], [53]. Our work complements bug localization techniques by providing an explanation of the bug. Not only ZIYUAN can benefit from better bug localization, but also the bug explanation identified by ZIYUAN could potentially help pinpoint where the bug is. This work is broadly related to research on the art of debugging, e.g., [39], [78], [48], [27], as well as recent studies on program repair, e.g., [50], [47], [52].

References

[1] http://javailp.sourceforge.net/.
[2] http://mernst.github.io/randoop/.
[3] http://sav.sutd.edu.sg/research/ziyuan/.
[4] https://code.google.com/archive/p/failuredoc/.
[5] https://code.google.com/archive/p/java-diff-utils/issues/10.
[6] https://code.google.com/archive/p/javaparser/issues/46.
[7] https://code.google.com/archive/p/javaparser/issues/57.
[8] https://github.com/hammacher/javaslicer.
[9] https://github.com/jodaorg/joda-time/issues/21.
[10] https://github.com/jodaorg/joda-time/issues/227.
[11] https://github.com/jodaorg/joda-time/issues/77.
[12] https://issues.apache.org/jira/browse/math-1005.

[13] https://issues.apache.org/jira/browse/math-1196.
[14] https://issues.apache.org/jira/browse/math-835.
[15] https://www.csie.ntu.edu.tw/~cjlin/libsvm/.
[16] http://www.eclemma.org/jacoco/.
[17] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. Spectrum-based multiple fault localization. In *ASE*, pages 88–99, 2009.
[18] A. Albarghouthi and K. L. McMillan. Beautiful interpolants. In *CAV*, pages 313–329, 2013.
[19] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of Interface Specifications for Java Classes. In *POPL*, pages 98–109, 2005.
[20] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. J. Trefler. Explaining counterexamples using causality. *Formal Methods in System Design*, 40(1):20–40, 2012.
[21] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *SIGSOFT/FSE'11*, pages 267–277, 2011.
[22] M. Botinčan and D. Babić. Sigma*: Symbolic Learning of Input-output Specifications. In *POPL*, pages 443–456, 2013.
[23] R. P. L. Buse and W. Weimer. Automatic documentation inference for exceptions. In *ISSTA*, pages 273–282, 2008.
[24] R. P. L. Buse and W. Weimer. Automatically documenting program changes. In *ASE*, pages 33–42, 2010.
[25] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.
[26] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.
[27] B. Cornu, E. Barr, L. Seinturier, and M. Monperrus. Casper: Debugging null dereferences with ghosts and causality traces. *CoRR*, abs/1502.02004, 2015.
[28] C. Csallner and Y. Smaragdakis. Dynamically discovering likely interface invariants. In *ICSE*, pages 861–864, 2006.
[29] G. de Caso, V. A. Braberman, D. Garbervetsky, and S. Uchitel. Automated abstractions for contract validation. *IEEE Trans. Software Eng.*, 38(1):141–162, 2012.
[30] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE*, pages 213–224, 1999.
[31] H. Estler, C. A. Furia, M. Nordio, M. Piccioni, and B. Meyer. Contracts in practice. In *FM*, pages 230–246, 2014.
[32] F. Fleurey, Y. L. Traon, and B. Baudry. From testing to diagnosis: An automated approach. In *ASE*, pages 306–309, 2004.
[33] G. Fraser and A. Arcuri. Evolutionary generation of whole test suites. In *QSIC*, pages 31–40, 2011.
[34] G. Fraser and A. Zeller. Generating parameterized unit tests. In *ISSTA*, pages 364–374, 2011.
[35] J. P. Galeotti, C. A. Furia, E. May, G. Fraser, and A. Zeller. Inferring loop invariants by mutation, dynamic analysis, and static checking. *IEEE Trans. Software Eng.*, 41(10):1019–1037, 2015.
[36] D. Giannakopoulou, Z. Rakamarić, and V. Raman. Symbolic Learning of Component Interfaces. In *SAS*, pages 248–264, 2012.
[37] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, pages 213–223, 2005.
[38] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN*, pages 121–135, 2003.
[39] Z. Gu, E. T. Barr, D. Schleck, and Z. Su. Reusing debugging knowledge via trace-based bug search. In *OOPSLA*, pages 927–942, 2012.
[40] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, pages 291–301, 2002.
[41] Z. Hassan, A. R. Bradley, and F. Somenzi. Incremental, inductive CTL model checking. In *CAV*, pages 532–547, 2012.
[42] F. Howar, D. Giannakopoulou, and Z. Rakamarić. Hybrid Learning: Interface Generation Through Static, Dynamic, and Symbolic Analysis. In *ISSTA*, pages 268–279, 2013.
[43] G. Hughes and T. Bultan. Interface grammars for modular software model checking. *IEEE Trans. Software Eng.*, 34(5):614–632, 2008.
[44] M. Jose and R. Majumdar. Bug-assist: Assisting fault localization in ANSI-C programs. In *CAV*, pages 504–509, 2011.
[45] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI*, pages 437–446, 2011.
[46] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*, pages 437–440, 2014.
[47] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search (T). In *ASE*, pages 295–306, 2015.
[48] D. Kim, X. Wang, S. Kim, A. Zeller, S. Cheung, and S. Park. Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *IEEE Trans. Software Eng.*, 37(3):430–447, 2011.
[49] S. Kim and E. J. W. Jr. How long did it take to fix bugs? In S. Diehl, H. C. Gall, and A. E. Hassan, editors, *MSR*, pages 173–174. ACM, 2006.
[50] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*, pages 3–13, 2012.
[51] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *SIGPLAN*, pages 141–154, 2003.
[52] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *POPL*, pages 298–312, 2016.
[53] A. T. Misirli, A. B. Bener, and B. Turhan. An industrial case study of classifier ensembles for locating software defects. *Software Quality Journal*, 19(3):515–536, 2011.
[54] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *ICST*, pages 153–162, 2014.
[55] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *ISSTA*, pages 229–239, 2002.
[56] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking. In *FSE*, pages 11–20, 2002.
[57] F. Orabona and N. Cesa-Bianchi. Better algorithms for selective sampling. In *ICML*, pages 433–440, 2011.
[58] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *ICSE*, pages 75–84, 2007.
[59] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, pages 199–209, 2011.
[60] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *FSE*, pages 23–32, 2004.
[61] L. H. Pham, L. L. T. Thi, and J. Sun. Assertion generation through active learning. In *ICFEM*, 2017. Accepted.
[62] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. *CoRR*, cs.SE/0310040, 2003.
[63] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. DARWIN: an approach to debugging evolving programs. *ACM Trans. Softw. Eng. Methodol.*, 21(3):19, 2012.
[64] D. Roach, H. Berghel, and J. R. Talburt. An interactive source commenter for prolog programs. In *SIGDOC*, pages 141–145, 1990.
[65] J. Rößler, G. Fraser, A. Zeller, and A. Orso. Isolating failure causes through test case generation. In *ISSTA*, pages 309–319, 2012.
[66] S. K. Sahoo, J. Criswell, C. Geigle, and V. S. Adve. Using likely invariants for automated software fault localization. In *ASPLOS*, pages 139–152, 2013.
[67] G. Schohn and D. Cohn. Less is more: Active learning with support vector machines. In *ICML*, pages 839–846, 2000.
[68] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as Classifiers. In *CAV*, pages 71–87, 2012.
[69] O. Strichman and M. Tautschnig. Learning the language of error. In *ATVA*, pages 114–130. Springer, 2015.
[70] J. Sun, H. Xiao, Y. Liu, S. Lin, and S. Qin. TLV: abstraction through testing, learning, and validation. In *ESEC/FSE*, pages 698–709, 2015.
[71] S. Tong and E. Y. Chang. Support vector machine active learning for image retrieval. In *ACM Multimedia*, pages 107–118, 2001.
[72] S. Tong and D. Koller. Support vector machine active learning with applications to text classification. *Journal of Machine Learning Research*, 2:45–66, 2001.
[73] W. E. Wong and V. Debroy. A survey of software fault localization. Technical Report UTDCS-45-09, 2009.
[74] H. Xiao, J. Sun, Y. Liu, S.-W. Lin, and C. Sun. TzuYu: Learning Stateful Typestates. In *ASE 2013*, pages 432–442, 2013.
[75] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu. Revisit of automatic debugging via human focus-tracking analysis. In *ICSE*, pages 808–819, 2016.
[76] A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE'99*, pages 253–267, 1999.
[77] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, pages 1–10, 2002.
[78] A. Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.
[79] S. Zhang, C. Zhang, and M. D. Ernst. Automated documentation inference to explain failed tests. In *ASE*, pages 63–72, 2011.