PathART: Path-Sensitive Adaptive Random Testing

Shan-Shan Hou¹,², Chun Zhang¹,², Dan Hao¹,², Lu Zhang¹,²
¹Key Laboratory of High Confidence Software Technologies(Peking University), MoE
²School of Electronics Engineering and Computer Science, Peking University, Beijing, China
{houss,zhangchun08,haod,zhanglu}@sei.pku.edu.cn

ABSTRACT
As test data widely spreading on the input domain may not thoroughly test the program’s logic, in this paper, we propose an approach to generating test data widely spreading on a program’s execution paths. In particular, we analyze execution paths of the program, distill constraints for executing the paths, calculate the path distance between test data according to their satisfaction for paths’ constraints, and then generate test data far away from each other based on their path distance. The experimental results show that our approach significantly reduces the number of test data generated before the first fault is found.

1. INTRODUCTION
Among various test-data generation techniques [5, 8, 10, 15], random testing attracts continuous research attention over the past decades due to its practical applicability and ease of implementation [9]. According to Ciupa et al. [4], a main weakness of random testing is that there is no systematic guidance of test-data selection, and thus various ways [2–4, 12] have been proposed to guide test-data generation in random testing.

Among these approaches, adaptive random testing (ART), which was first proposed by Chen et al. [2, 3], has attracted much research attention. The goal of ART is to select a set of randomly generated test data that are as far away from each other as possible. The intuition behind this goal is that, for a set of test data close to each other, when one test data in the set is unable to reveal any fault, it is likely that the other test data in the set are unable to reveal any fault either. In particular, ART generates candidate test data randomly, calculates the distances between each candidate test data and all the existing test data, and chooses the candidate test data that is farthest away from existing test data as the next test data.

In ART, the metric used to measure the distance between two test data plays an important role. In fact, existing research on ART mainly focuses on the definition and calculation of distances between test data. For example, the distance metric in ART by localization [3] and ART through dynamic partitioning [2] is the Euclidean distance, and the distance metric in ART0O [4] is the combination of the elementary distance, the type distance, and the filed distance. All these existing metrics are based on only the values of the two test data. However, for a particular program under test, two test data that are far from each other under an existing metric may actually execute similar paths or even the same path in the program under test. In other words, two test data with very different values may turn out to be close to each other from the perspective of the program under test.

To further consider information of the program under test in ART, we propose to use constraints for paths of the program under test to measure the distances between test data. Thus, our approach to ART is sensitive to paths of the program under test. To evaluate our approach, we performed an experimental study of purely random testing, ART using the distance metric defined by Ciupa et al. [4], and our approach on 25 subjects. The experimental results show that our approach is able to generate much fewer test data than the other two approaches for revealing the first fault in a subject averagely, although our approach requires the most analysis time to generate one test data.

2. PATHART
In this paper, we propose an approach to ART named PathART, in which we use paths of the program under test to measure the distance between test data. Similar to existing approaches to ART [2–4], PathART uses an iterative procedure to acquire a set of test data. When acquiring each test data, PathART first generates some candidate test data, and then selects one test data based on the distances between each candidate test data and all the test data acquired previously. PathART stops when a fault is revealed or the number of generated test data reaches a given value. PathART consists of four main steps to acquire one test data as follows.

First, select a subset of program paths based on the distances between test data in terms of paths of the program under test.
Second, calculate the constraint of the path in the form of input variables by applying symbolic execution along the path.
Third, randomly generate a set of candidate test data, some of which come from the data pool and some of which are acquired from the input domain.
Fourth, use path constraints to measure distances between test data and select one test data from the set of candidate test data.

Noted that PathART needs to do program-path selection, path-constraint construction, and data-pool construction only once and uses the path constraints and the data pool for all the subsequent candidate test-data generation and test-data selection. In the following sections, we present the details of these four steps.

2.1 Selecting Program Paths
The basic idea of our PathART is measuring data distances according to their execution paths of the program under test. Therefore, we must construct a set of execution paths of the program...
under test. In our PathART, we consider to select a subset that can pair-wisely cover the programs’ branches, denoted as pw-subset.

We use pw-subset for the following reasons. First, a pw-subset is subset of all execution paths; using such subset can help avoid the overwhelming cost of enumerating all execution paths. Second, we can estimate the scale of pw-subset with the help of current techniques. As we generate and select test data according to these execution paths, too many execution paths may induce high computation cost.

Pair-wise coverage is widely used in combinatorial test-data generation for testing software systems [7]. In particular, if the system under test has many parameters and each parameter has only two optional values, the system is a system with many 2-level factors [11]. Researchers have proposed various algorithms of combinatorial test-data generation for systems with many 2-level factors [7, 11].

In our PathART, we reduce a switch branch as nested if branches, and reduce a loop to nested if branches by unfolding the loop body a certain number of times. Therefore, any branch node in a CFG has only two different branches. Viewing each branch node with two branches as a parameter and each branch as an optional value of the parameter, the pw-subset can be summarized as a system with many 2-level factors. Therefore, we can use their proposed algorithms [7, 11] to select a pw-subset of program paths for the program under test in the following way.

First, for a CFG of the program under test, we select all sequential branch nodes, which are branch nodes that are not nested in other branch nodes. For the preceding sequential branch nodes, we generate branch values for them to satisfy pair-wise coverage of these branches.

Second, if a branch node has nested branch nodes, we recursively generate branch values to pair-wisely cover these nested branch nodes. Then we combine each nested branch value with its nesting branch value of sequential branch nodes to obtain a group of execution paths.

We process nested branch nodes recursively, until all nested branch nodes have been given values and combined with their nesting nodes. That is, we acquire a pw-subset of execution paths of the program under test.

### 2.2 Constructing Path Constraints

For measuring the distance between test data and each data candidate based on the differences of their execution paths, we estimate their execution paths without executing the data candidate. In our PathART, we estimate a data candidate’s execution path by observing its satisfaction to a program path’s constraint.

To acquire path constraints of selected program paths, we implement forward symbolic execution [6] along all the selected program paths, and represent the branch predicates in the form of input variables. Definition 1 formally defines the path constraint of a given path.

**Definition 1. Path Constraint.** For a given path \( p=(N_1(X), ..., N_m(X)) \), where \( N_i (1 \leq i \leq m) \) is a branch node and \( X \in \{0, 1\} \), the path constraint of \( p \) is in its conjunctive normal form if and only if it is in the form \( C_p=\bigwedge_{i=1}^m c_i \) \((1 \leq i \leq m)\), where \( c_i \) is the branch condition of \( N_i(X) \).

In fact, as there are black-box function invocations, branch conditions may not be induced as formulae with input variables, and the symbolic execution may be interrupted. For example, if a local variable in a branch condition is the return value of a function of a reusable API library whose source code is not available, the local variable cannot be represented with input variables. During forward symbolic execution, in this situation, our PathART stops symbolic execution and uses the current constraints collected so far as the path constraint.

### 2.3 Generating Candidate Test Data

Some recent research [4] has shown that test-data generation based on data pools can improve the effectiveness of pure random testing. In other words, when generating a value for a parameter, we randomly select a value from the input domain with a probability \( p \), and randomly select a value from the data pool with a probability \( 1-p \). In our PathART, we construct data pools containing not only special values of the data type, such as 0, 1, and -1 for integer, but also boundary values of program paths’ constraints.

For a branch condition \( X \in \Theta \) in a program path’s constraint, where \( X \) is an input variable of a numeric type, \( \Theta \) is a relational operator, and \( V \) is a value of the numeric type, values \( V, V-1, \) and \( V+1 \) are also added into the data pool. When generating data candidates of a primitive data type, we randomly select a value from the input domain of the data type with a probability \( p \), and randomly select a value from the data pool with a probability \( 1-p \).

### 2.4 Selecting Test Data

#### 2.4.1 Path Distance

As mentioned earlier, our PathART defines data distances according to the differences of their execution paths. We first define a data’s satisfaction degree to a path \( p \) in Definition 2, and then define the distance between two test data according to their satisfaction degrees to a set of program paths.

**Definition 2. Satisfaction Degree for a Path.** Given path \( p \) whose path constraint is \( C_p=\bigwedge_{i=1}^m c_i \) \((1 \leq i \leq m)\), where \( c_i \) is the branch condition of \( N_i(X) \), the test data \( t \), its satisfaction degree for \( p \) is \( S(p, t)=\frac{j-1}{m+1} \cdot 100\% \), where \( j \) is the index of the branch condition \( c_j \) \((1 \leq j \leq m+1)\), which is the first branch condition evaluated as “false” using \( t \). Note that if all branch conditions are evaluated as “true”, the value of \( j \) is \( m+1 \).

**Definition 3. Path Distance.** Given a program path set \( P \) containing \( n \) program paths (denoted as \( P=\{p_1, ..., p_n\} \)), a test data \( t \) and a candidate data \( t_c \), the path distance between \( t \) and \( t_c \) is \( D_{dist}(t, t_c, P)=\sum_{i=1}^n D_{satisfaction}(S(p_i, t), S(p_i, t_c)) \), where \( D_{satisfaction}(S(p_i, t), S(p_i, t_c)) \) are the satisfaction degrees of \( t \) and \( t_c \) for path \( p_i \) in \( P \), respectively.

#### 2.4.2 Test-Data Selection

After calculating the distances between test data and each candidate data, to distribute test data on different program paths, our PathART chooses the test candidate data that is farthest away from existing test data as the next test data. The number of existing test data is usually more than one, for a candidate test data, we calculate the average path distance between the candidate test data and all the existing test data. The candidate test data with the maximal average path distance is the one that is the farthest away from all the existing test data. Figure 1 shows the algorithm TD-Selection on computing the average path distance. For each candidate test data, TD-Selection first calculates the path distance between the candidate test data and each of the existing test data, then computes the average path distance between the candidate test data and all the existing test data, and finally selects the candidate test data with the maximal average path distance as the next test data.

### 3. EXPERIMENTAL STUDY

In the experimental study, we applied our PathART with pure random (in short as Random) and classical ART by localization [3] (in short as ART) on 25 real small programs, which are the solution
to some problems in the ACM International Collegiate Programming Contest (in short as ACM-ICPC). Peking University holds an online judge library (in short as POJ)\(^1\) for ACM-ICPC, which contains ACM-ICPC problems and their solutions submitted by the students of Peking University.

For each ACM-ICPC problem, users of POJ (mainly students of Peking University) submitted a large number of programs, some of which contain real faults. In this experimental study, we chose 5 ACM-ICPC problems (i.e., Biorhythms Problem\(^2\), Packing Problem\(^3\), Chocolate Problem\(^4\), Multiply Problem\(^5\), and Josephus’ Problem\(^6\)) from POJ. For each of the preceding 5 problem, we randomly select 5 Java programs with faults. As POJ provides sample solutions for each problem, the sample solutions serve as test oracles in our study. For each problem, we label its five subjects as A, B, C, D, and E, respectively.

### 3.1 Tools, Algorithms, and Parameters

To analyze the programs under test, we adopt the Joeq compiler\(^7\) to construct CFGs for the programs under test. Also, we implement forward symbolic execution on the quads code of the programs under test; such quads code is an output of Joeq compiler.

Based on the CFG generated by Joeq, we compute the post-dominant set of each node, and use the pairwise-covering algorithm of test-data generation for the system with many 2-level factors \(^11\) to select key program paths in our study.

When generating data candidates, we set the probability \(p\) for selecting data from input domain as 0.25, which is a good value according to previous research on random test-data generation \(4\).

### 3.2 Threats to Validity

In our experimental study, there are two main threats to external validity. The first threat to external validity lies in the subjects and faults. Although faults and programs are similar to real because they are collected from the ACM-ICPC online judge library, all the programs are small Java programs whose faults may not represent real faults in software development. To reduce this threat, we plan to evaluate our approach using other real larger programs in other languages with real faults in future work. The second threat to external validity lies in different probability values of \(p\) that control the probability of selecting data from data pools and input domains, which may induce different experimental results. In our study, we use a probability value 0.25 following other researchers’ work, in which 0.25 is reported as a good value. To reduce this threat, we plan to use different probability values in future evaluation of our approach.

### 3.3 Results and Analysis

During the iterative process of test-data generation using Random, ART, and PathART, the test data generation is stopped if a real fault is revealed or the number of the generated test data achieves 1000. As all the Random, ART, and PathART are based on random test-data generation, we generate test data for each program 100 times, and report the average values of them in this study. The experimental subjects and results are available on our project webpage\(^8\).

We first compare the effectiveness of revealing faults for Random, ART, and PathART. As mentioned earlier, for each problem, we have five faulty programs submitted by different students. Table 1 illustrates fault exposure by the test data generated using Random, ART, and PathART. From the table we can see that there are cases in which our PathART finds faults that Random and ART do not find with 1000 test data. In summary, the test data generated by PathART reveal all the 25 faults, whereas the test data generated by either Random or ART reveal 22 faults.

Table 2 shows the number of the test data required to reveal the first fault of the program under test. As Random and ART cannot reveal faults for three subjects of the chocolate problem, the corresponding entries are filled with dashes. From Table 2, our PathART outperforms ART and Random on 23 subjects (except for C and D of Josephus’ Problem) substantially.

We further study two subjects on which ART outperforms Random and PathART. We find that the programs of these two subjects consist of many loops. In our PathART, we deal with loops by unfolding them a certain number of times. In this study, we unfold loops 0, 1, 2, and 3 times to acquire key program paths. We suspect the reason of the preceding phenomenon to be that the dealing of loops induces inaccurate path-analysis results, which cannot effectively guide test-data generation.

From Table 2, we observe that ART cannot outperform Random. In some cases, the number of test data required for revealing the first fault of ART is smaller than that of Random, such as the subject A of the biorhythms problem. However, in other cases, the number of test data required for revealing the first fault of Random is much smaller than that of ART, such as the subject D of the packing problem. We suspect the reason is that test data are transformed by the logics of the program under test, and thus the strategy of ART, which aims at distributing test data widely on the input domain, cannot guide the test-data generation effectively.

We performed the study on a 2.5G Hz Intel Core 2 Duo with 4GB RAM system running Mac OS X 10.5.6, recording the time of our PathART on analyzing paths and generating test data. We

---

\(^1\)http://poj.grids.cn/  
\(^2\)http://poj.grids.cn/problem?id=1006  
\(^3\)http://poj.grids.cn/problem?id=1017  
\(^4\)http://poj.grids.cn/problem?id=1322  
\(^5\)http://poj.grids.cn/problem?id=1331  
\(^6\)http://poj.grids.cn/problem?id=3254  
\(^7\)http://suif.stanford.edu/courses/cs243/joeq/index.html  
\(^8\)https://sites.google.com/site/asergrp/projects/pathart
found that the maximum time cost of our PathART for all the 25 programs is 5.688 seconds, and the maximum average time cost of our PathART for all the five problems is 1.533 seconds. That is, the time cost of our PathART is acceptable.

4. RELATED WORK

Our work is related to random testing, which generates test data for programs randomly. Previous research [9] has evaluated the effectiveness and cost of random testing. With the ease of implementation in testing tools, some popular testing tools, such as JCrasher [1] and Randoop [12], have implemented techniques of random test-data generation. Recent research improves the effectiveness of random testing using feedback information of test execution [12]. Other research improves the effectiveness of random testing by guiding the distribution of test data [2–4]. Different from existing work, our PathART improves the effectiveness of random testing based on the program-path information and aims to generate test data evenly distributed on different execution paths of the program under test.

Further, our work is also related to test-data generation based on path constraints, which generates test data executing a given path of the program under test [14]. These approaches usually use a technique such as symbolic evaluation to acquire accurate constraints of executing the given path, and generate test data for that path by solving the constraints. Some recent research [13] has developed concolic testing techniques, which concretely execute programs together with symbolic execution of the executed path. Different from existing work, our PathART does not solve path constraints, but randomly generates test data and evaluates their deviation from the path constraints.

5. CONCLUSION

In this paper, we have proposed a novel technique to generate test data widely spreading on a program’s execution paths. In particular, we select a group of key execution paths of the program under test, analyze constraints for executing the paths, generate data candidates, calculate the path distance between existing test data and each data candidate according to their satisfaction for path constraints, and select the data candidate that is farthest away from each other as the next test data according to the path distance. The experimental results show our approach is effective on revealing real faults of the program under test with acceptable time cost.

6. ACKNOWLEDGMENTS

This research is sponsored by the National Basic Research Program of China (973) under Grant No. 2009CB320703, the High-Tech Research and Development Program of China under Grant No. 2011AA01A202, and the Key Program of Ministry of Education, China under Grant No.313004, and the National Natural Science Foundation of China under Grant Nos. 61121063, 60803012, and 61272157.

7. REFERENCES