A Heuristic Test Data Generation Approach for Program Fault Localization

Saeed Parsa, Hamid Rafati PorShokooh, Saman Teymouri,
and Mojtaba Vahidi-Asl

Department of Software Engineering,
Iran University of Science and Technology, Tehran, Iran
{parsa,m_vahidi_asl}@iust.ac.ir,
{h_rafati,steymouri}@comp.iust.ac.ir

Abstract. The aim of this paper is to improve the reliability of programs by generating test cases considering different execution paths of the program. The method introduced in this paper assumes that only a single failing run is available for the program and applies a genetic algorithm which searches for the most similar failing and passing runs in terms of their executed predicates. By contrasting the similar passing and failing runs, the predicates that are different in two executions could be reported as fault relevant ones. We have also applied the *k-means* clustering technique to partition test cases according to their corresponding execution paths in order to ensure about the quality of software and locate the existing faults of the program. To evaluate the accuracy of the proposed method, we have conducted some case studies on a number of Siemens programs including different faulty versions. The results show the capability of the proposed method in generating a wide variety of test cases which could cover different program execution paths. The results also show the effectiveness of the approach in localizing faults according to detected fault relevant predicates.

Keywords: Software Testing, Fault Localization, Test Case Generation, Genetic Algorithm.

1 Introduction

The major goal of software companies is to produce reliable bug free products. To ensure the reliability and quality of software, a comprehensive testing process should be done before the deployment phase. Software testing is the process of examining software product in pre-defined ways to check whether the behavior is according to the expected behavior. Typically, one objective of software testing is designing minimal number of test cases such that they manifest as many faults as possible [15][16].

In this paper, we attempt to integrate the process of software testing with fault localization. Two well-known fault localization techniques are statistical debugging [2] and program slicing [3]. Statistical debugging techniques rely on a huge number of failing and passing test cases which are often collected from end users as error

H. Deng et al. (Eds.): AICI 2011, Part I, LNAI 7002, pp. 236–243, 2011.

reports. These reports are analyzed using statistical methods to construct statistical model of program behavior that is used to pinpoint the location of faults [1].However, for many applications we have no access to user reports (i.e., a collection of failing and passing test cases) and we only have got one failing test case. In these situations, program slicing techniques could be applied. Among different variations of program slicing techniques, backward dynamic slicing computes all executed statements having influence on a specific execution point [4]. However, dynamic slices could be large and hence considerable human effort might be needed to find the faulty code.

For many applications, we merely have a single failing run and we have two important concerns. Firstly, how we could generate test cases in order to ensure the reliability of the software. Secondly, how we could generate a passing test case for which the dynamic behavior of the program is the most similar with its behavior for the existing failing test case. By finding the most similar passing to the failing test case and contrast their corresponding dynamic behaviors, we could report the differences as the fault suspicious code that is responsible for the program failure [6]. This paper attempts to address the mentioned problems. Assuming that there is only one failing test case available, namely the starting test case, we slightly alter the existing test case by considering *m*-combinations of input parameters (*m*=1..*n*, *n* is the number of input parameters) to generate new test cases. To be more precisely, we first select a one parameter subset and change the corresponding values (i.e. $\binom{n}{1}$), then subsets of two parameters (i.e. $\binom{n}{2}$), until all the *n* elements are changed (i.e. $\binom{n}{n}$).During the process of test case generation, we may also find other failing test cases and hence the proposed approach could detect more than a single bug in the program. this process could be known as heuristic test data generation.

The proposed approach has two significant consequences: (1) Testing different paths of program (2) Program bug localization. The problem of generating the most similar passing to failing test cases in terms of their corresponding execution paths is known to be NP-complete [9] and there is no effective way to find the solution. Thus, we try to solve the problem by formulating it as a search problem and applying genetic algorithms to find the best solution.

The remaining part of this paper is organized as follows. In section two, related works in the context of test case generation is presented. Section three describes the proposed method in detail. The experimental results are shown in section four. Finally, conclusions and future works are mentioned in section five.

2 Related Work

Different approaches have been proposed for test case generation [18]. They could be roughly categorized to random, intelligent (i.e. heuristic), path based and goal based techniques. The random techniques generate a huge amount of test data in a blind manner. However, in these techniques the testing requirements are not considered in generating test data and a big proportion of data might be redundant and useless.

Intelligent techniques on the other side perform some analysis during generation of test data which makes the technique more expensive when a huge amount of analysis is required for specific applications. The path based techniques generate test cases

which could traverse particular paths of a given program. In goal based techniques the aim is to generate inputs which could traverse paths among two specific points of the program [10].

Recently, different approaches for test case generation using genetic algorithms have been introduced [7][11]. Due to the complexity of software testing problems, genetic algorithms could be best applied to search for optimal (or near-optimal) solutions. The approach presented by Pargaset. al. [11] uses genetic algorithm to automate the process of goal based test data generation. Given particular goals like specific statements, branches, paths or definition-use pair in a program under test, it seeks for a test data which could cover the desired goals.

Girgiset. al. [12] has proposed a technique that uses GA guided by the data flow dependencies in the program to search for test data which fulfills data flow path selection criteria, namely the all-uses criterion. Singla's and Rai presented anautomatic test-data generation technique that uses a particle swarm optimization (PSO) to generate test data that satisfy data-flow coverage criteria[13].

The idea of comparing similar failing and passing runs to find the differences as the cause of failure was first purposed by [6]. However, the method assumes that a large number of failing and passing test cases are available and among these test cases it finds the most similar ones to contrast. Therefore, it does not attempt to minimize the difference between failing and passing runs. For some programs, the difference might be too large which confuses the programmer to locate the origin of failure. In contrast, our purposed method assumes that only single faulty path is available and tries to find a test case corresponding to the most similar passing one.

3 An Overview of the Method

The proposed method has three main stages: 1) Test case generation 2) clustering the execution traces 3) Narrowing down the passing and failing traces using genetic algorithm. Each main phase is described in detail in the following sub-sections.

3.1 Test Data Generation

Before describing the process of test case generation, we first define the failing and passing test cases.

Definition 1. Given program P and a set of test cases, Tc_P: each test case, tc_j consists of input parameters, I_j, and the corresponding desired output, O_j. After executing P on tc_j,$Exp_P(tc_j)$, the output result would be $\acute{O}_j$: $Exp_P(tc_j) = \acute{O}_j$. Test case tc_j is called failing test case, ${tc_j}^{fail}$, if: $\acute{O}_j \neq O_j$ and otherwise it is named passing test case, ${tc_j}^{pass}$. Therefore, the set of test cases in Tc_P is split into two disjoint categories: ${Tc_P}^{Pass}$ for passing test cases and ${Tc_P}^{fail}$ for failing test cases.

In this paper, we assume that there is only one failing test case which is considered as our initial point to generate other test cases. We name this significant test case, tc_{start}^{fail}.

To generate test cases, we first study the data structure of the available failure inducing input. Then we decompose tc_{start}^{fail} to its parameters. In next step, in each iteration of the proposed algorithm, a subset of input parameters, (i.e. *m*-combinations of the input) and their corresponding values are changed in order to generate new test cases. With this slight variation, we try not to deviate too much from the existing faulty path corresponding to tc_{start}^{fail}.

To record the execution path that is generated by running the program with a particular test data, it is required to insert some probes before specific points of the program. These specific points, namely predicates, could be the decision making statements (e.g. conditions, loops, function calls, etc.) where the execution path of the given program is determined. The process is known as instrumentation.

Definition 2. Running the instrumented program P with test case tc_j generates an execution path, $Epath_P^j$, which is a sequence of executed predicates (i.e. predicates that are evaluated as TRUE). The execution vector is a binary vector $Evec_P^j = (X_1, X_2, \dots, X_n)$ in program execution pace, where X_i is the value of predicate x_i in that run (i.e. '1' if the predicate is evaluated as True and '0', otherwise).

In remaining parts of the paper when we talk about the distance between two execution paths A and B we actually address the difference in their corresponding execution vectors, $Evec_P^A$ and $Evec_P^B$. It is also important to label each execution vector as fail/pass according to the program termination status in that particular run.

3.2 Clustering the Execution Vectors

By mapping each particular run to an adequate vector, now we can use Euclidean distance to measure the similarity of executions in terms of their execution vectors. It is clear that execution vectors with more common features (i.e. executed predicates) may share the more execution sub-paths and vice versa.

Definition 3. Given program P, The similarity of two test cases tc_m and tc_n , $\Delta(tc_m, tc_n)$, is defined in terms of the Euclidean distance between their corresponding execution vectors, $Evec_P^m$ and $Evec_P^n$.

To identify different execution paths, we use the K-means clustering technique [14] on our failing and passing execution vectors. This strategy helps us to make our test data richer by considering the clusters (i.e. execution paths) in isolation.

Definition 4. Given a set of test cases for program $P(tc_1, tc_2, tc_3, \dots, tc_m)$, without regarding the fail/pass label of the test cases, where each test case is represented as *n*-dimensional vector, $Evec^i$, *k*-means clustering aims to partition the *m* test cases into *k* sets ($k \leq m$) CL = {cl_1, cl_2, …, cl_k} according to the following within-cluster function which minimizes the sum of squared Euclidean distances:

$$\underset{CL}{arg\min} \sum_{i=1}^{k} \sum_{tc_j \in cl_i} \|tc_j - c_i\|^2 \quad (1)$$

Where c_i is the center of execution vectors in the cluster cl_i.

3.3 Applying a Genetic Algorithm to Generate Similar Paths

A typical Genetic algorithm starts with an initial population including a set of solutions so called individuals. Each individual, also known as a chromosome, is conventionally a string of bits associated with a fitness value that determines the probability of survival of an individual chromosome in the next generation [7][17]. In our application each chromosome is the string of input parameters. To evolve the initial population into a new generation there are four significant mechanisms: 1) initialization 2) selection 3) crossover and mutation 4) termination. Each mentioned stage is described in detail in the following sub-sections.

Initialization and Selection. The set of input test parameters corresponding to execution vectors in each cluster is considered as the initial population for the genetic algorithm. For each cluster, at first place, the corresponding population is explored to find the most proper failing test case. The adequacy is determined according to the distance of the corresponding execution failing vector with the corresponding passing vectors in the neighborhood. The corresponding test case is considered as the pattern of our genetic algorithm. The fitness function computes the Euclidean distance between the execution vector of the pattern and the execution vector of the generated test case.

Mutation and Cross over operations. To generate a new population of test cases, two major operators are used, cross over and mutation. The crossover operation involves two chromosomes (i.e. parents) exchanges substring information at random position in the string to produce two new strings (i.e. children). A predefined percentage of population is selected as parents in each iteration to form the new population. In this paper we have used one point cross over technique. To be more precise, the population individuals are sorted according to their corresponding fitness values. Then, for 30% of individuals from the bottom of the population (i.e. the solutions with the least fitness values) we have applied the cross over operation. The reason for choosing the test cases with large Euclidean distance from the candidate failing test case is to evolve the solutions as much as possible. At this point, the chosen test cases are paired to form parents and for each pair we do as follows. Assuming that each solution (i.e. parent) consists of n parameters, a number, namely i, is randomly selected in the range of $[0,n]$ and parameters from i to n are swapped in two given parents. The new children are evaluated by the fitness function and replaced by their corresponding parents.

To perform mutation, a predefined amount of the population (i.e. 20% in our application) is randomly selected. After that, in each selected individual an input parameter (i.e. gene) is chosen randomly and the corresponding value is replaced with some other value in the range of the parameter. The new individual is evaluated by the fitness function and being replaced by its parent if the corresponding fitness value is improved after the modification of the parameter.

Termination. The termination condition is if the successive iterations no longer produce better results. This is controlled by a time threshold in order to avoid the infinite loop problem. The final population contains test cases which are sorted according to the distance to the candidate test case. Therefore, the passing test case with the highest priority is our desired solution. Now, it is only required to contrast

both paths to find the differences. The differences are predicates which could be suspicious for the failure of the program.

The genetic algorithm is applied on all clusters which specify different execution paths. With this strategy, we try not to miss the faults which appear in different paths of the program. Furthermore, it is possible that the genetic algorithm in one cluster could generate better results in comparison with other clusters.

4 Experimental Results

In this section we show the effectiveness of applying genetic algorithm on generating a passing test case which has the least distance with the starting failing test case. Fig. 2 shows the results of applying genetic algorithm to reduce the distance in the first version of Print-Tokens program. In the Figure, the trend of genetic algorithm in different clusters for reducing the distance between starting failing test case and the generated passing one in version 1 of the program is presented. Fig. 2 compares the genetic and the heuristic method (i.e. *m*-combination changes of the parameters to generate new test cases) in reducing the distance in version 3 of the same program. Fig. 3 shows that in some situations, the heuristic test case generation method has generated the passing test case with the least distance and therefore there is no need for applying the genetic algorithm. This shows that for some cases, the heuristic method is adequate by itself to find the cause of failure. The overall evaluation of the method on some versions of Tcas, schedule and Print-Tokens programs is presented in table 1.

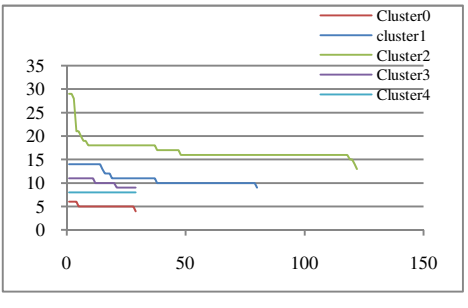

Fig. 1. The Trend of Genetic algorithm in reducing the distance between faulty and passing execution in different clusters in version 1 of Print-Tokens

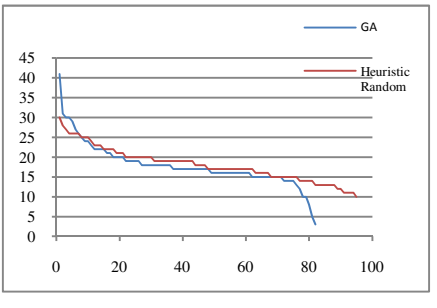

Fig. 2. The comparison of Genetic and heuristic method in reducing the distance in version 3 of Print-Tokens

5 Concluding Remarks

In this paper, a new method has been introduced for testing and bug localization in programs. We start from one single failing run to generate new passing and failing test cases in a heuristic manner. Using a genetic algorithm, we try to generate passing test cases for which their corresponding execution vectors have the least difference

with the available failing test case. The results show the capability of our proposed method in both generating test cases for testing different paths of a program and finding fault relevant predicates to locate the origin of failure. For many versions, the exact location of failure is included in the difference set of predicates resulted from comparing similar passing and failing runs. For other versions, by a little examination of code we could find the origin of failure according to the reported fault relevant predicates. By applying the genetic algorithm on different test cases without regarding its failing or passing state, we could considerably increase the path coverage of the test cases.

However, some issues should be considered for our future work. We may introduce a ranking model to prioritize predicates according to their fault relevance. We will attempt to evaluate the method on larger programs with more complex structure. We would like to evaluate the method in circumstances that instead of a failing test case, there is only single passing one. Improving the genetic algorithm by considering the probability distributions of the predicates may also be studied for our future work.

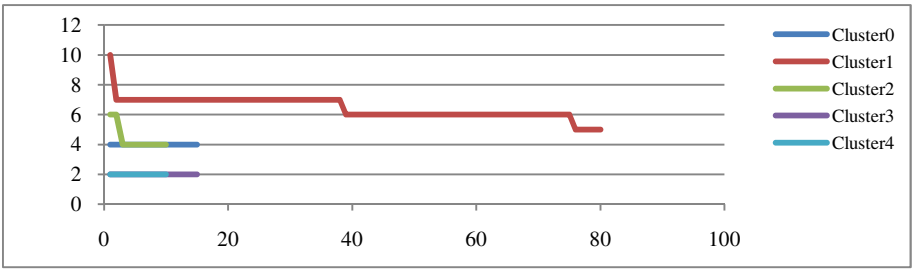

Fig. 3. The capability of the heuristic method in finding very similar passing and failing test cases without the need for applying the genetic algorithm

Table 1. The overall evaluation of the proposed method on some versions of tcas, schedule and Print-Tokens programs

program	Cluster	Optimum cluster	Best Initial distance	Final distance	Distance between failure origin And reported predicates (LOC)
Print-Tokens (V1)	5	0	6	4	0
Print-Tokens (V2)	5	3	12	10	1
Print-Tokens (V3)	5	3,4	2	2	0
Print-Tokens (V5)	5	0	1	1	0
Print-Tokens (V7)	6	3	7	2	0
Schedule (V1)	5	0	7	3	2
Schedule (V4)	2	0	4	2	2
Schedule (V8)	5	0	3	1	3
Tcas (V1)	2	1	3	2	0
Tcas (V3)	2	1	3	3	0
Tcas (v4)	2	0	3	3	0
Tcas (V5)	2	0	3	2	0
Tcas (V6)	2	1	2	2	0

References

1. Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.: SOBER: Statistical model-based bug localization. In: Proceedings of the 10th European Software Engineering Conference and 13th ACMSIGSOFT International Symposium on Foundations of Software Engineering, pp. 286–295. ACM Press, New York (2005)
2. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 141–154. ACM Press, New York (2003)
3. Zhang, X., Gupta, R., Zhang, Y.: Precise dynamic slicing algorithms. In: IEEE/ACM International Conference on Software Engineering (ICSE), Portland, pp. 319–329 (2003)
4. Agrawal, H., De Millo, R., Spafford, E.: Debugging with dynamic slicing and backtracking. Software Practice and Experience (SP&E) 23(6), 589–616 (1993)
5. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann, San Francisco (2006)
6. Renieris, M., Reiss, S.: Fault localization with nearest neighbor queries. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, pp. 30–39 (2003)
7. Mantere, T., Alander, J.T.: Evolutionary software engineering, a review. Journal of Applied Soft Computing Archive 5(3), 315–331 (2005)
8. Software Infrastructure Repository, `http://www.cse.unl.edu/_galileo/sir`
9. Lei, Y., Tai, K.: In-parameter-order: a test generation strategy for pairwise testing. In: Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium, Washington DC, pp. 254–261 (1998)
10. Korel, B.: Automated software test data generation. IEEE Transactions on Software Engineering (1990)
11. Pargas, R.P., Harrold, M.J., Robert, R.P.: Test-data generation using genetic algorithms. In: Software Testing, Verification and Reliability (1999)
12. Girgis, M.R.: Automatic test data generation for dataflow testing using genetic algorithm. Journal of Universal Computer Science 11(6), 898–915 (2005)
13. Singla, S., Singla, P., Rai, H.M.: An Automatic Test Data Generation for Data Flow Coverage Using Soft Computing Approach. International Journal of Research and Reviews in Computer Science (IJRRCS) 2 (2011)
14. Tan, P., Steinbach, M., Kumar, V.: Introduction to data mining. Pearson, Addison Wesley (2006)
15. Desikan, S., Ramesh, G.: Software testing Principles & Practices. Pearson Education, London (2007)
16. Ammann, P., Offutt, J.: Introduction to software testing. Cambridge University Press, Cambridge (2008)
17. Sivanandam, S.N., Deepa, S.N.: Introduction to Genetic Algorithms. Springer, Heidelberg (2010)
18. Edvardsson, J.: A survey on automatic test data generation. In: Proceedings of the Second Conference on Computer Science and Engineering in Link, pp. 21–28 (1999)