An Application of Program Slicing in Evolutionary Testing

Yongping Zhang Jianhua Sun Shujuan Jiang
School of Computer Science and Technology
China University of Mining and Technology,
Xuzhou, China, 221116
shjjiang66@163.com

Abstract—**Evolutionary Testing is a kind of efficient method of automatically test data generation. It uses a kind of meta-heuristic search technique, the Genetic Algorithm, to convert the task of test data generation into an optimal problem. One problem for evolutionary testing is the convergence speed will be degression when a problem has many variables and is within a large input domain. In this paper, we suggest that program slicing can be used to narrow the search space, and the test data generation framework that applies the method is given, and then a case study is presented in the end.**

Keywords-Program slicing; evolutionary testing; test data generation; path testing; software testing.

I. INTRODUCTION

Evolutionary Testing (ET) is a search-based software engineering technique, based upon evolutionary algorithms [1,2]. ET utilizes a fitness function to evaluate an individual in an original population, and generates the offspring population from some genetic operation. Under the guide of the function, after several iterative generations, we could get the global optimal solution [3]. Genetic algorithms (GA) are a group of evolutionary algorithms, that use the evolution principle that was originally proposed by Charles Darwin[4-5].

Recent researches have covered many areas. Michael and McGraw [6] have developed the so-called genetic algorithm data generation tool (GADGET) system that is fully automatic and supports all C/C++ constructs. The system is used to obtain condition/decision coverage. They also examine the effect of program complexity on the test data generation problem. Bueno and Jino [7] have studied the possibility of using a GA to identify the potentially infeasible program paths. They proposed that monitoring the progress of the GA search could identify an infeasible path. Thesis by Tracey [8] deals with automatic test data generation for testing safety critical systems. He uses simulated annealing and genetic algorithms, but also random search and hill climbing as the optimization methods. It is observed that ‘‘genetic algorithm-based approaches for structural test data generation have a number of weaknesses that restricts their application to real software industry.

Although evolutionary testing works well in many situations, the convergence speed will be degression when a problem has many variables and a large input domain. In this paper, we present a method that program slicing is applied to reduce search space size and give a framework of generating test data by slicing and genetic algorithm. The approach can reduce the time of generating test data and improve the test efficiency.

The rest of the sections are organized as follows. Section 2 introduces the problem for evolutionary testing. Section 3 describes how to reduce search space size by slicing. The construction method of fitness function is given in Section 4. Section 5 presents a framework of generating testing data by slicing and genetic algorithm. Section 6 presents a case study and section 7 concludes the paper.

II. PROBLEM FOR EVOLUTIONARY TESTING

Evolutionary testing may not be the final answer to the software testing problem, but do provide an effective strategy. Inevitably large and more complex search spaces will reduce the effectiveness and efficiency of a search based approach [6,8]. We also find that the convergence speed will be degression when a problem has many variables and within a large input domain. Program slicing [9,10] can be used to reduce the search space size, by focusing attention on a particular “slice” of the source code. Slices are constructed using dependence analysis. Linear dependence level reductions cause exponential search space reductions [11].

If we can delete the statements that are not related to the test requirements by slicing, the search space size will reduce and the testing efficiency will improve.

III. SLICING TO REDUCE SEARCH SPACE SIZE

Search algorithms are particularly sensitive to the size of the search space. The size of the search space is exponential in the number of input variables to the program, so methods which reduce this size may produce exponential speed ups in the search [12].

Program slicing is an effective technique for narrowing the focus of attention to the relevant parts of a program. A slice consists of statements and predicates that have influence on the variables at a program point. There are two kinds of slicing methods: static slicing and dynamic slicing. Static slicing

Supported by National Natural Science Foundation (No.60970032), the Key Project of Chinese Ministry of Education (No.108063), Natural Science Foundation of Jiangsu Province (No. BK2008124). Science Research Foundation of China University of Mining and Technology (0D080310)

978-1-4244-4994-1/09/$25.00 ©2009 IEEE

computes slices by taking all the possible executions into consideration, and dynamic slicing is with respect to the given input sets. Static slicing is mainly used in program understanding and maintenance because the information is global. A dynamic slice contains fewer statements than the static one, thus it is useful for debugging and localizing errors [13,14].

In this paper, we focus our attention on the static slicing methods. Every program slice is based on a rule, named as slicing criterion. In static slicing, the slicing criterion has the form <i, v> where i is the serial number of a statement in a program point, v is variable set. The difference between our approach and conventional method is the slicing criterion. We must define the slicing criterion according to test requirement, according to the test coverage requirement for what is "interesting". The slicing criterion used in our approach is the form <i, x>, where x denotes the variable set, i is different from the previous slicing criterion. It is not the serial number of a statement, but the line-number of a statement in a program.

Slicing on the Program Dependence Graph (PDG) of sequential programs is a simple graph reachability problem, for control and data dependence is transitive [10,15].

IV. FITNESS FUNCTION CONSTRUCTION

The key problem in evolutionary testing is constructing the fitness function to evaluate the testing data. In this paper, we use the Korel's branch function. The fitness function of Korel's branch function is :

Assuming the branch predication *pr* is a relational expression, E_1 OP E_2, where E_1 and E_2 is arithmetic expression, relational operator OP$\in\{<, >, \leq, \geq, =, \neq\}$. We convert *pr* to a predication style, F rel 0, where rel$\in\{<, \leq, =\}$. The F is Korel's branch function. The method is simple and executable, but it only can generate a testing data for one path by running one time. If we need more testing data for different paths, we need change the function more times.

The original population is generated by random, $G_m(x_1,x_2,\ldots,x_n)$. Assuming matrix M[r,c], where r is the number of path, c=m and the initial value is 0. Let each individual $G_i(x_1,x_2,\ldots,x_n)$ (i=1,2,…,m) to execute the tested program. If G_i exercises the path r, the value of M[r,i] is assigned to 1. Let

$$F_i = \sum_{j=1}^{m} M[x, j] \quad (1)$$

Where x is the row whose i-th column is 1 in the matrix. The fitness function is the following.

$$\text{fitness}(i) = C - F_i \quad (2)$$

where C is a constant. When the testing data for a path is generated many times, F_i will increase, and then the value of fitness function will decrease. This can encourage to generate testing data for others paths.

When selecting the offspring population, we scan each row of matrix M and find the first element, M[i,j], whose value is 1, then the individual j enters the offspring population.

The algorithm termination condition is that at least an element 's value is 1 in all the paths in matrix M or the iterative times reach 1000 generations.

V. A FRAMEWORK OF GENERATING TEST DATA

Program slicing can improve the drawback of convergence speed degression because of many variables and the large search space. The system for generation of test data consists of three parts: pre-process part, slicing part and test data generation part, as shown in Figure1.

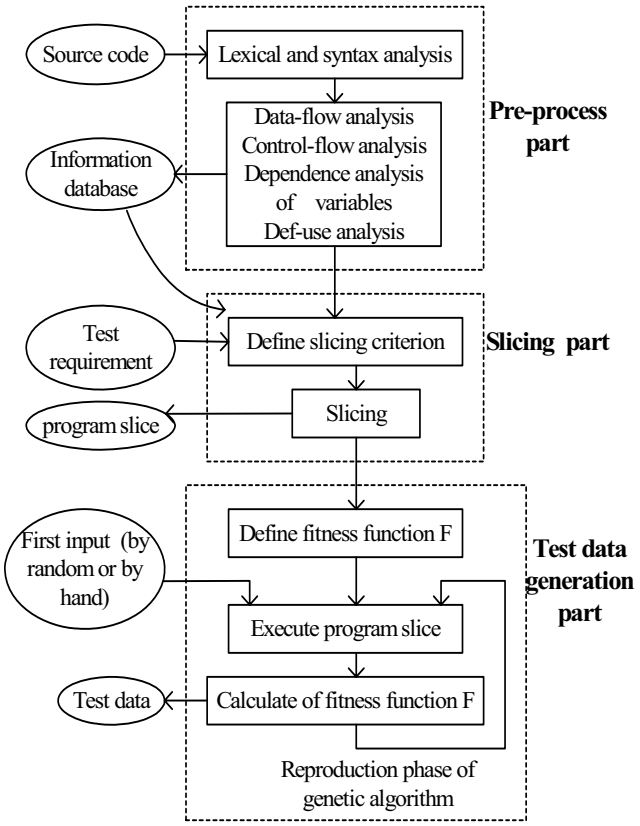

Figure 1. The framework of generating test data

Firstly, we do Lexical and syntax analysis on source code, and analyze the control-flow, data-flow of the program, and also the dependency relationship of the variables. The analysis results are stored in a public information database.

Secondly, we define the slicing criterion according to the test requirement and the information in the public information database. The program is sliced and the program slice is obtained.

Thirdly, We define the fitness function F of program slice based on test requirement, then execute program slice under the first input that is obtained by random or by hand and calculate the fitness function F. The genetic search process is iterative: evaluating, selecting, and recombining strings in the

population during each iteration (generation) until reaching some termination condition occurs, such as a success leads to termination of the search, as does a protracted failure to make any forward progress.

If the termination of the search is due to a success, the input is the test data that satisfy the test requirements.

VI. A CASE STUDY

In this section, we present an empirical study we performed to evaluate our technique. The example is a program that determines whether it is an equilateral triangle from input data. The source code of the program is shown in figure 2.

```
1  int a,b,c;
2  char  *t=(char *)malloc(50*sizeof(char));
3  scanf("%d,%d,%d",&a,&b,&c);
4  if(a<=0||b<=0||c<=0)
5     t="Input error!";
6  else
7  {    if(a+b<=c||b+c<=a||a+c<=b)
8            t="not a triangle";
9        else
10       {    if(a==c&&c==b)
11               t="equilateral triangle";
12           else {
13           if((a==b&&a!=c)||(a==c&&b!=c)||(b==c&&b!=a))
14               t="isosceles triangle";
15           else
16               t="general triangle";
17               }
18       }
19 }
```

Figure 2 The source code of the example

If we want to generate the testing data for equilateral triangle, firstly, we need to do the static slice under the slicing criterion <11,t> . It is shown in figure 3. Secondly, we generate the testing data using the genetic algorithm. The fitness function is the formula that we describe in section 4. The algorithm termination condition is at least an element ‘s value is 1 in the row that corresponding to the path of equilateral triangle in matrix M or the evolutionary searches are terminated after 1000 generations.

The table 1 is the contrast of the evolution testing data generations between not applying slice technique and applying slice technique. Here, the times denote the i-th execution program, such as 2 expresses the second executing program. The number expresses the iterative times, such as 719 means that it obtains the test case after 719 generations. Where 1000+ expresses that evolutionary search is terminated after 1000 generations of no improvement in the best fitness value.

```
1  int a,b,c;
2  char  *t=(char *)malloc(50*sizeof(char));
3  scanf("%d,%d,%d",&a,&b,&c);
4  if(a<=0||b<=0||c<=0)
5     t="Input error!";
6  else
7  {    if(a+b<=c||b+c<=a||a+c<=b)
8            t="not a triangle";
9        else
10       {    if(a==c&&c==b)
11               t="equilateral triangle";
12       }
13 }
```

Figure 3 The static slice of the example

According to Table 1, we can conclude that the average iterative times after slicing is less than one that not applying slicing, so the efficiency of testing data generate is improved if we combine slicing and genetic algorithm.

TABLE 1 THE CONTRAST OF THE ITERATIVE TIME BETWEEN NOT APPLYING SLICING AND APPLYING SLICING

Times	**1**	**2**	**3**	**4**	**5**	**6**	**Average**
Before slicing	719	406	448	624	1000+	697	649
After slicing	149	163	37	588	249	57	207.2

VII. CONCLUSION

In this paper, we suggest that slicing can be used to narrow the search space in evolutionary testing. We also give the generating test data framework that applies the method. From the initial experiments in structural testing we discovered that the approach could improve the testing efficiency. The advantages of the method include two points.

The program slicing can reduce the search space size. This can overcome the limitation that the convergence speed is degression when a problem has many variables and within a large input domain.

The search process is iterative and it needs to execute the program many times. In our method, the execute program is the program slice rather than the source code. This can save the execute time because the program slice is only one part of the source code, which can really improve software testing efficiency.

[1] M. Harman and B. F. Jones. Search based software engineering. Information and Software Technology, 43(14):833–839, Dec. 2001.

[2] A. Baresel, D. Binkley, M. Harman, et al. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. ISSTA’04,July 11-14,2004, Boston, Massachusetts, USA.

[3] H.H. Sthamer. The automatic generation of software test data using genetic algorithm. PhD Thesis, University of Glamorgan, Pontyprid, Wales, Great Britain, April 1996.

[4] J.H. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, 1975.

[5] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, 1989.

[6] C.C. Michael, G.E. McGraw, M.A. Schatz, Generating software test data by evolution, IEEE Trans. Software Eng. 27 (12) (2001) 1085–1110.

[7] P.M.S. Bueno, M. Jino. Identification of potentially infeasible program paths by monitoring the search for test data, in: Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering, ASE 2000, IEEE Piscataway, NJ, Grenoble, France, 2000, pp.209–218.

[8] N. Tracey. A search-based automated test-data generation framework for safety-critical software, Ph.D. thesis, University of York, 2000.

[9] M. Weiser Program Slicing, IEEE Trans. Software Engineering, 1984, 16(5): 498-509.

[10] S. Horwitz, et al. Interprocedural Slicing Using Dependency Graphs. ACM Trans. Programming Languages and Systems, 1990,12(1): 26-60

[11] D. Binkley, M. Harman. Analysis and visualization of predicate dependence on formal parameters and global variables. IEEE Transaction on software engineering, 30(11):715-735,2004.

[12] M. Harman, L. Hu, R. Hierons, et al. Evolutionary testing supported by slicing and transformation. In: Proceedings of the International Conference on Software Maintenance(ICSM/02).

[13] J. Zhao Dymanic Slicing of Object-Oriented Programs. Technical-Report SE-98-119, Information Processing Society of Japan, May 1998 : 17-23

[14] L.D. Larsen, M.J. Harrold. Slicing Object-Oriented Software, Proc. 18th Int'l Conf. Software Eng., pp.495-505, 1996

[15] Z.Q. Chen, B.W. Xu. Dependence analysis based dynamic slicing for debugging. Wuhan University Journal of Natural Science, 2001,6(1-2):389-404.