An Approach on Automatic Test Data Generation with Predicate Constraint Solving Technique

Jifeng Chen[1], Li Zhu[2], Junyi Shen[1], Zhihai Wang[1], and Xinjun Wang[1]
[1] Institute of Computer Software, Xi'an Jiaotong University, 710049 Xi'an, China
{jfchen, jyshen}@mail.xjtu.edu.cn
[2] School of Software, Xi'an Jiaotong University, Xi'an 710049, China
zhuli@mail.xjtu.edu.cn

Abstract

Predicate constraint solving technique is an important method of automatic test data generation. By analyzing the properties and disadvantages of predicate constraint solving technique, three theorems are proposed and proved. Based on them, a new approach on automatic test data generation is presented. The linear predicate on a given path is used directly to construct the linear constraint system. Only the linear arithmetic representation for nonlinear predicate function is required to compute. Theoretical analysis and practical testing show: for the linear path, the initial input and the iteration are not required. The path is feasible if the linear constraint system can be solved, and the values of the input variables obtained are the desired test data. Otherwise, the path is infeasible. Considering the nonlinear path, test data can also be obtained by a number of iterations, or it can be guaranteed that the path is infeasible to a great extent.
Keywords: predicate constrain solving techniques; linear constrain; linear arithmetic representation

1 Introduction

Software testing is one of the critical activities during software development. It is often claimed that testing and debugging account for approximately 50% of software development costs, which may be thought as a sign of the need for automated testing support[1]. Software testing automation can relieve software engineers from their tedious daily task, and reduce the cost of software developing significantly. Several approaches have been proposed for automated test data generating, including random method[2], syntax based method[3] , program specification based method[4], symbolic evaluation method [5,6] and program execution based method[7-11]. Predicate constraint solving techniques are widely used for automatically test data generating[6,8,11-20]. This paper presents a novel test data generating approach, using predicate constraint solving techniques for program execution, and describes its algorithm in detail, and then illustrates it by two examples.

Foundation item: Project (2003AA1Z2610) supported by National Hi-Tech Research and Development Program of China.

This paper is organized as follows. Next section explores the related works, which lead to our new test data generating approach. Section 3 details the algorithm of our approach, and illustrates it with examples involving linear and nonlinear paths in section 4. Finally, in Section 5, we summarize the important features of our approach.

2 Related Work

One of the earliest systems using symbolic evaluation only for linear path constraints to generate test data automatically is described in [5], which is able to detect infeasible paths with linear path constraints except its limitation of handling array reference which depended on program input. A more recent attempt of using symbolic evaluation to generate test data for fault based criteria is described in [6]. In this work, a test data generation system based on a collection of heuristics for solving system of constraints is developed. The derived constraints are often imprecise, resulting in an approximate solution on which the path may not be traversed. Since the test data is not refined further to obtain the desired input eventually, the method will fail when the path is not traversed on the approximate solution. The approaches in [7,8] use backtracking manner and considered only one branch predicate and one input variable each time. Therefore, they may require a large number of iterations even if all the branch conditionals along the path are linear functions of the input. If more than one conditions on the selected path depended on common input variables, a lot of effort will be wasted in backtracking. They also can not consider all the branch predicates on the path simultaneously because the path may not be traversed on an intermediated input.

In order to solve the problems above, a method, proposed in [19], replaces predicate function with linear arithmetic representation and established linear constraint system of predicate functions on the increments for the input. The test data is obtained through iterative refining in this method. The number of program execution in each iteration is independent of the path length and at most equaled to the number of input variables plus one. Therefore, the waste in backtracking in [7,8] can be avoided. At the same time, it is needed to compute the slices, the linear arithmetic representation, and the predicate residuals of all predicate functions, to identify the input dependency sets and to construct the linear constraint system of predicate functions on the increments for the input. Hence, the method is also complex, and is then further simplified in [20], which gets rid of the need of both computing the slices and identifying the input dependency sets, but still need to compute the linear arithmetic representation and the predicate residuals of all predicate functions.

In our new approach, the linear predicate function is used directly to construct the linear constraint of predicate function of the input. There is no need to compute the linear arithmetic representation of linear predicate function, except that the predicate function is nonlinear. Only the predicate functions on the input variables need to establish the linear constraint system, but the predicate functions on the increments for the input do not need. In addition, there is no need to compute the slices and iden-

tify the input dependency sets. So, this is a much simpler approach and will cost less in computing than the all previous since a large part of predicate functions are linear in common programs.

3 Algorithm

3.1 Backgrounds

For the sake of simplicity, some basic concepts of the approach are defined using the program as follows:

```
Program 1
   0:    read (X,Y,Z)
   1:    U=(X-Y)*2
   P1:   if (X>Y) then
   2:          W=U
   3:    else W=Y
         endif
   P2:   if (W+Z)>100 then
   4:          X=X-2
   5:          Y=Y+W
   6:          write ("Linear")
   P3:   elseif (X^2+Z^2>100) then
   7:          Y=X*Z+1
   8:          write ("Nonlinear: Quadratic")
         endif
   P4:   if (U≥0) then
   9:          write (U)
   P5:   elseif (Y-Sin(Z))>0 then
   10:         write ("Nonlinear: Sine")
         endif
```

Definition 1: A conditional expression in multi-way decision statement on path is called a **Branch Predicate**, such as $X>Y$ in $P1$ of program1.

Definition 2: Each branch predicate $E1$ op $E2$ can be transformed to the equivalent branch predicate of the form F op 0, Where $E1$, $E2$ and F are arithmetic expressions, $F= E1-E2$ and op is one of $\{<, \leqslant, >, \geqslant, =, \neq\}$. Along a given path, F represents a real valued function called a **Predicate Function**. For example, branch predicate $X>Y$ can be transformed to $X-Y>0$, then $F=X-Y$ is called predicate function.

Definition 3: A general linear function $L(n_i, I_k, P)$ is written according to the input variables of predicate node n_i for the given input I_k on path P. Then, the values of the coefficients in the general linear function are computed so that $L(n_i, I_k, P)=0$ repre-

sents the tangent plane to the predicate function at I_k. Therefore, $L(n_i, I_k, P)$ is called the **Linear Arithmetic Representation** of predicate function F in predicate node n_i for the given input I_k on path P.

Definition 4: If predicate function F is linear, expression F op 0 is called its linear constraint of the predicate function. Otherwise, linear arithmetic representation $L(n_i, I_k, P)$ is called linear constraint of the predicate function.

3.2 Mathematic Foundation

In this section, three theorems based on iterative relaxation method [19, 20] are presented, and the corresponding proofs are given below.

Theorem 1 If the predicate function on a given path is linear, then its linear arithmetic representation is itself.

Proof Since the predicate function is linear, the predicate function can be described as follows:

$$F(X) = a_1X_1 + a_2X_2 + \ldots + a_iX_i + \ldots + a_mX_m + a_0$$

- X_i represents input variable; m is the number of input variables, $i \in \{1,2,\ldots,m\}$.
- a_i represents the coefficient corresponding to input variable.
- a_0 represents constant.

Let k represent iterative number, $k \in \{0,1,2,\ldots,T\}$, then the divided difference of the k^{th} iterative $F(X)$ on $X_{i,k}$ is:

$$F[X_{i,k+1}, X_{i,k}] = \frac{F(X_{i,k+1}) - F(X_{i,k})}{X_{i,k+1} - X_{i,k}} = \frac{a_{i,k}(X_{i,k+1} - X_{i,k})}{X_{i,k+1} - X_{i,k}} = a_{i,k}$$

The derivative of $F(X)$ on $X_{i,k}$ is: $F'_{X_i} = a_{i,k}$

Obviously, the coefficient corresponding to input variable obtained with divided differences is equivalent to the first derivative of $F(X)$, and is unrelated to the input variable. Therefore, the linear arithmetic representation of the linear predicate function is itself.

Theorem 2 The linear constraint of predicate function for the current input is equivalent to that on the increments for the input.

Proof Let the predicate function be as follows (the parameters representation are the same as that in theorem 1):

$$F(X) = a_1X_1 + a_2X_2 + \ldots + a_iX_i + \ldots + a_mX_m + a_0 = \sum_{i=1}^{m} a_i X_i + a_0$$

According to iterative relaxation algorithm, the predicate residual of the k^{th} iterative $F(X)$ on $X_{i,k}$ is

$$R(X_{i,k}) = \sum_{i=1}^{m} a_{i,k} X_{i,k} + a_{0,k}$$

The linear constraint representation of the k^{th} iterative $F(X)$ on the increments for the input is

$$\sum_{i=1}^{m} a_{\mathrm{i,k}}\ \Delta X_{\mathrm{i,k}}\ + R(X_{i,k})\ \mathrm{op}\ \ 0 \qquad (1)$$

where op$\in \{<, \leqslant, >, \geqslant, =, \neq\}$

$$\because\ \Delta X_{i,k} = X_{i,k+1} - X_{i,k}\ ,$$

Take $\Delta X_{i,k}$ and $R(X_{i,k})$ into (1), then we get

$$\sum_{i=1}^{m} a_{i,k}\ X_{i,k+1}\ +\ a_{0,k}\ \ \mathrm{op}\ \ \ 0 \qquad (2)$$

Equation (2) is the linear constraint representation of predicate function $F(X)$ on input variables. $X_{i,k+1}$ is the input variable value I_{k+1} of the $(k+1)^{th}$ iteration. As the result of (2) obtained from (1), (2) is equivalent to (1). Hence, theorem 2 is proved.

Theorem 3 If all branch predicate functions on a given path P are linear, then path P is feasible while the linear constraint system can be solved, and the values of the input variables obtained are the desired test data. Otherwise, path P is infeasible.

Proof Let us assume that there are m input variables for the program containing the given path P and there are n branch predicates on the path P. Each branch predicate can be transformed to one of the forms $\{F=0, F>0, F\geqslant 0\}$. Assume $n1$ of them use "=", $n2$ use ">", and $n3$ use "$\geqslant$". Then $n=n1+n2+n3$. According to the given conditions and previous two theorems, the set of linear constraints for all branch predicates on path P can be constructed as below:

$$\begin{cases} AX + A_0 = 0 \\ BX + B_0 > 0 \\ CX + C_0 \geq 0 \end{cases} \qquad (3)$$

Where $A = \begin{pmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,m} \\ \ldots & \ldots & \ldots & \ldots \\ a_{n1,1} & a_{n1,2} & \ldots & a_{n1,m} \end{pmatrix}$, $B = \begin{pmatrix} b_{1,1} & b_{1,2} & \ldots & b_{1,m} \\ \ldots & \ldots & \ldots & \ldots \\ b_{n2,1} & b_{n2,2} & \ldots & b_{n2,m} \end{pmatrix}$,

$C = \begin{pmatrix} c_{1,1} & c_{1,2} & \ldots & c_{1,m} \\ \ldots & \ldots & \ldots & \ldots \\ c_{n3,1} & c_{n3,2} & \ldots & c_{n3,m} \end{pmatrix}$, $X=(x_1, x_2, \ldots, x_m)^T$, $A_0=(a_{1,0}, a_{2,0}, \ldots, a_{n1,0})^T$,

$B_0=(b_{1,0}, b_{2,0}, \ldots, b_{n2,0})^T$, $C_0=(c_{1,0}, c_{2,0}, \ldots, c_{n3,0})^T$

$\because$ "$\geqslant$"is equivalent to ">"$\bigcup$"="

$\therefore$ (3) can be separated into two sets of linear constraints:

$$\begin{cases} AX + A_0 = 0 \\ BX + B_0 > 0 \\ CX + C_0 > 0 \end{cases} \quad (4), \qquad \begin{cases} AX + A_0 = 0 \\ BX + B_0 > 0 \\ CX + C_0 = 0 \end{cases} \quad (5)$$

Both (4) and (5) are the set of constraints which is only composed of "=" and ">". While solving the set of constraints above, (4) and (5) are either consistent or inconsistent. There is a set of test data which can traverse path *P* if at least one of (4) and (5) is consistent. The solution is the desired test data. Otherwise, Path *P* is infeasible if both (4) and (5) are inconsistent.

3.3 Algorithm Description

According to the theorems in section 2, the main idea of the new approach is: checking each branch predicate on a given path, if all the branch predicates are linear expression, algorithm 1 is executed; otherwise algorithm 2.

Algorithm 1: All the branch predicate functions are used directly to construct the linear constraint system for the current input variables. Further, the linear equation system for the input variables is established and solved. Solution is the desired test data for input variables. If the constraint system is inconsistent, then the path is guaranteed to be infeasible.

Algorithm 2: Choosing a set of input variable values in given domain to check each branch predicate on the path, the linear arithmetic representation of nonlinear predicate functions on current input is computed. The linear constraint system on the input variables is constructed with the linear predicate functions on the path and the linear arithmetic representations obtained previously. Further, the linear equation system on the input variables is established, and solved to get the values of input variables. Hence, a set of new input is obtained. If the set of new input cannot traverse the given path, then the process above is repeated till the desired outcome is obtained or the iterative upper limit is achieved.

4 Application

Our approach is illustrated to solve the linear constraint system with two examples involving linear and nonlinear path separately.

Example 1 the path with linear predicate function

Program 1 is used to analyze the automatic test data generation of linear predicate functions on a given path.

Suppose the path *P* is selected:

P={0,1, *P*1,2, *P*2,4, 5,6,*P*4,9}

According to the path *P*, the algorithm is executed as follows:

Algorithm 1 is executed due to all predicate functions (*P*1, *P*2, *P*4) on *P* are linear.

Construct the linear constraint system of predicate functions directly：

$$\begin{cases} X - Y > 0 \\ 2X - 2Y + Z - 100 > 0 \\ 2X - 2Y \geq 0 \end{cases}$$

Solve the linear constraint system.

The above inequalities are converted to equalities by introducing three new variables *a, b, c.* where *a, b* >0 and *c*≥0. The set of equalities is below：

$$\begin{cases} X - Y = a \\ 2X - 2Y + Z - 100 = b \\ 2X - 2Y = c \end{cases}$$

The values of free variables can be selected arbitrarily with the constraints that *a, b* >0 and *c*≥0. The values of free variables *a, b, c* are chosen as 1. Solving the equality system according to the method in [17], the result is X=2.6，Y= 2，Z=99.8. Hence, the input I=(2.6, 2, 99.8) is the desired test data.

If choose the path P={0,1, P1,2, P2,4, 5,6,P4, P5,10}, the set of linear constraints can be obtained:

$$\begin{cases} X - Y > 0 \\ 2X - 2Y + Z - 100 > 0 \\ 2X - 2Y < 0 \\ Y - \sin(Z) > 0 \end{cases}$$

Obviously, the linear constraints, X-Y>0 and 2X-2Y<0, are inconsistent. So the constraint system cannot be solved. With theorem 3, it is guaranteed that this path must be infeasible. It is easy to check that P is indeed an infeasible path.

Example 2 the path with nonlinear predicate function

For the sake of simplicity, the program follows is used to generate test data for the path with nonlinear predicate function.

```
Program 2
    1:    float  x;
    2:    scanf("x",&x);
    P1:   if (x<-1);
    P2:           if (x*x)>0;
    3:                  print ("Ok");
    4:            else print ("No");
    5:    else print ("No");
```

Choosing and checking path P = {1,2, P1, P2, 3}, Algorithm 2 is executed as predicate function P2 on P is nonlinear. Given any arbitrarily chosen input I_0 in the program domain and iterative increments of input variables ΔX. e.g. $I_0 = X_0 = 1$, $\Delta X = 1$.

As the path P is not traversed on $I_{0,}$ the steps for iterative refinement of I_0 are executed.

Step 1 is executed since predicate function $P2$ is nonlinear.

Step 1 Compute the linear arithmetic representation of $P2$

Since the predicate function of $P2$ is $F=X^2$, the linear arithmetic representation can be represented as follows:

$$L(BP2, I_k, P)=aX+b$$

Approximate the derivatives of a predicate function by its divided differences, then a=3, and

$$3X_0+b= X_0^2$$

Solve the equation, b=-2. The linear arithmetic representation of predicate function $F= X^2$ is

$$L(BP2, I_0, P)=3X-2$$

Step 2 Construct the linear constraint system of predicate function on I_0 using linear predicate functions $P1$ and the linear arithmetic representation $L(BP2, I_0, P)$

$$\begin{cases} X+1<0 \\ 3X-2>0 \end{cases}$$

Step 3 Solve the linear constraint system

Using the solving method of linear constraint system in example 1, the solution is X=0.7. Therefore, the new input I_1=0.7.

step1 ~ step3 for iterative refinement I_1 is needed to executed repeatedly since the path P is not traversed on I_1 yet. Finally, the path P is traversed on I_6=-1.3867 obtained in the 6th iteration. Therefore, the algorithm is determined. I_6 is the desired test data. The results of execution of our test data generation approach for this example in the table are given below.

Table 1 Test data and the coverage of Branch Predicates(BP) on each iteration

Iteration	*X*	*BP1*	*BP2*
0	1	F	T
1	0.7	F	T
2	0.4817	F	T
3	0.2810	F	T
4	0.0361	F	T
5	-0.4125	F	T
6	-1.8135	T	T

Where F and T represent False and True respectively.

5 Conclusions

While using the new approach to derive the desired test data for a given path, the linear constraint system of predicate functions for the input variables is directly constructed if all the predicate functions on the path are linear. The initial input and the iteration are not needed. Either the desired test data is derived or it is guaranteed that the path is infeasible from the solution of the constraint system. The new approach need not compute the predicate residuals and the linear arithmetic representation of linear predicate function. The constructions of predicate slice, input dependency set and the linear constraint of predicate function on the increments for the input can also be omitted. Only when the predicate function is nonlinear, the linear arithmetic representation needs to be computed.

The experiments were executed on a Pentium-based computer 1.6G with 512MB RMA running Linux OS (Red Flag 4.1). Each program is executed for 5 times. For the program 1, the average executing time of test case generation of method [19], method [20] and our method are 34ms, 26ms and 22ms respectively, and are 28ms, 24ms, and 20ms respectively for program 2. Theoretical analysis and practical testing results show that the approach is simpler, easier, more effective, and less cost in computing than the others. However, the test data can't be always guaranteed to generate if there are one or more nonlinear predicate functions on the path.

References

1 Korel B. *Automated software test data generation*. IEEE Transactions on Software Engineering, 1990, 16(8): 870-879.
2 Avritzer A, Weyuker E. J. *The automatic generation of load test suites and the assessment of the resulting software*. IEEE Transactions on Software Engineering, 1995, 21(9): 705-716.
3 Deason W.H, Brown D.B, Chang K, Cross J.H. *A rule based software test data generator*. IEEE Transactions Knowledge and Data Engineering, 1991, 3(1): 108-116.
4 Bauer J, Finger A. *Test plan generation using formal grammars.* Proceedings of the 4th International Conferene of Software Engineering. Munich, Germany, 1979. 425-432.
5 Clarke L. A. *A system to generate test data and symbolically execute programs*. IEEE Transactions on Software Engineering, 1976, 16(8): 870-879.
6 DeMillo R, Offutt J. *Constraint-Based Automatic Test Data Generation* . IEEE Transactions on Software Engineering, 1991, 17(9): 900-910.
7 Gallagher M. J, and Narasimhan VL. ADTEST: *A Test Data Generation Suite for Ada Software Systems*. IEEE Transactions on Software Engineering, 1997, 23(8): 473-484.
8 Gotlieb A, Botella B, M. Rueher. *Automatic test data generation using constraint solving techniques* . In Proceedings of the Sigsoft International Symposium on Software Testing and Analysis . Florida, USA, 1998. 53- 62.
9 Mansour N, Salame M, Joumaa R. *Integer- and real-value test generation for path coverage using a genetic algorithm*. In Software Engineering and Applications Conference (SEA). 2000, 49-54.
10 Gupta N, Mathur A. P, and Soffa M. L. *Generating test data for branch coverage*. In 15th IEEE International Conference on Automated Software Engineering (ASE00). Grenoble, France, 2000.

11 Beydeda S, Gruhn V. *Test Data Generation based on Binary Search for Class-level Testing*. ACS/IEEE International Conference on Computer Systems and Applications. Tunis, Tunisia, 2003. 107-114
12 Gotlieb A, Botella B, Rueher M. *A CLP framework for computing structural test data*. Proceedings of the First International Conference on Computational Logic. London, 2000. 399–413.
13 Michel C, Rueher M, and Lebbah Y. *Solving constraint over floating-point numbers*. In Seventh International Conference on Principles and Practice of Constraint. Paphos, Cyprus, 2001.
14 TranSy N, Deville Y. *Consistency Techniques for Interprocedural Test Data Generation*. ACM SIGSOFT Software Engineering Notes archive ,2003.
15 Gupta N, Cho Y-J, Hossain M. Z. *Experiments with UNA for Solving Linear Constraints in Real Variables.* Proceedings of the 2004 ACM symposium on Applied computing. Nicosia, Cyprus, 2004.1013-1020.
16 Gupta N, Cho Y-J, Hossain M. Z. *UNA: A Simple Numerical Algorithm to Solve Linear Constraints in Real Variables*. Technical Report TR 03-08, Computer. Science Department, The University of Arizona, 2003.
17 Gupta N, Mathur A. P, Soffa M. L. *UNA Based Iterative Test Data Generation and its Evaluation*. 14th IEEE International Conference on Automated Software Engineering (ASE'99). Florida, 1999. 224-232.
18 Edvardsson J, Kamkar M. *Analysis of the Constraint Solver in UNA Based Test Data Generation*. Foundations of Software Engineering Proceedings of the 8th European software engineering conference. Vienna, Austria,2001. 237 – 245.
19 Gupta N, Mathur A. P, Soffa M. L. *Automated Test Data Generation Using An Iterative Relaxation Method*. Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Florida, United States,1998. 231 - 244.
20 SHAN Jin-Hui. *On Path-wise Automated Test Data Generation*. Changsha: National University of Defense Technology, 2002.

Jifeng CHEN was born on August 1, 1966. He is currently working towards the Ph.D degree in computer science and technology, Institute of Computer Software and Theory, Xi'an Jiaotong University, China. His research interests are in the general field of software testing and software engineering.

Li ZHU, PH.D, associate professor, was born on January 28, 1967. He received his Doctor degree in computer science & technology from Xi'an Jiaotong University in October 2000. His main research interests are software testing, novel networking technology, software engineering and embedded system and so on.