Domain Testing Based on Character String Predicate

Ruilian Zhao
Computer Science Dept.
Beijing University of
Chemical Technology

Michael R. Lyu
Computer Science Dept.
Chinese University of
Hong Kong

Yinghua Min
Institute of Computing Tech.
Chinese Academy of Sciences
in Beijing

Abstract

Domain testing is a well-known software testing technique. Although research tasks have been initiated in domain testing, automatic test data generation based on character string predicates has not yet been reported. This paper presents a novel approach to automatically generate ON-OFF test points for character string predicate borders associated with program paths, and describes a corresponding test data generator. Slices with respect to predicates on paths are constructed to calculate the current values of variables in the predicates via program slicing techniques. Each character element of variables in a character string predicate is dynamically determined in turn by function minimization so that the ON-OFF test points for the predicate border can be automatically generated. The preliminary experimental results show that this approach is promising and effective.

1. Introduction

Software testing is an important stage to guarantee software quality and reliability [1]. At present, a large number of software testing approaches have been developed to detect either data flow or control flow errors [2]. Domain testing is a well-known control flow based testing technique, which attempts to reveal errors in the predicates that affect the flow of control through a program by selecting test points on and near the boundary of a path domain. Although several research activities have been initiated in domain testing [3, 4, 5, 6], automatic test data generation based on character string predicate has never been reported. All recent domain testing strategies are limited to programs in which each predicate can contain Boolean variables, relational expressions or Boolean operators, but character string variables are not allowed. This severely restricts domain testing strategies for application in practice since character string predicates are widely used in modern programming techniques.

In this paper, we present a novel approach to automatically generate *ON-OFF* test points for character string predicate borders associated with program paths, and develop a corresponding test data generator. Instead of using symbolic execution or program instrumentation, we construct a slice with respect to a predicate on a path via program slicing techniques. The current values of variables in the predicate are calculated by executing the slice, thus avoiding the problems found in symbolic execution and the costly and time-consuming jobs for designing proper instrumentation statements. Each element of variables in a character string predicate is determined in turn by performing function minimization so that the *ON-OFF* test points for the corresponding predicate border are automatically generated. Our preliminary experimental results show that this approach is effective and promising.

The remainder of this paper is organized as follows. Section 2 introduces domain testing strategies. Section 3 reviews briefly dynamic test data generation. Section 4 describes main principle of automatic *ON-OFF* test point generation for character string predicate borders, and provides a corresponding test generator. Section 5 reports an example study to indicate that the test generator is practical. Finally, conclusion is presented in Section 6.

2. Domain testing strategies

Program errors can be classified into two categories: computation errors and domain errors [7]. A program is said to cause a *computation error* if a specific input follows a correct path, but the output is incorrect due to faults in some computations along the path. A *domain error,* which can be manifested by a shift in some segment (border) of the path domain boundary, occurs when a specific input traverses a wrong path because of faults in the control flows of the program.

To detect domain errors or provide confidence in the correctness of path domain, White and Cohen proposed a domain testing strategy [3]. For a linear predicate with a total of *n* distinct arithmetic variable, the strategy requests to design *n ON* test points and one *OFF* test points. These *ON* test points lie on the border to be tested, while the *OFF* test point is placed slightly off the border on the outside, and is close to these *ON* test points. Zeil et al extended domain testing to detect linear errors in a nonlinear predicate [4]. Afterwards, Jeng and Weyuker [5] presented a simplified domain testing strategy, which requests to generate one *ON* and one *OFF* test points in any dimension for an inequality border that the corresponding predicate contains operator ≤, <, ≥ or >. For an equality or non-equality border associated with

operator $=$ or $\neq$, one *ON* test point and two *OFF* test points are required. The only prescription is that *ON* test point should lie on the border whereas *OFF* test point should be placed outside the border, and the *ON-OFF* point pair has to be as close to each other as possible. Furthermore, Hajnal and Forgacs [6] introduced an algorithm to generate *ON-OFF* test points by the simplified domain testing strategy with some manual assistance. All these domain testing strategies, however, suffer from a common weakness: character string predicates are not taken into account.

3. Dynamic test data generation

Dynamic test data generation is the most often used approach for developing test data. In this paper, we employ dynamic test data generation to derive *ON-OFF* test points for character string predicate borders associated with a path.

As discussed in [8], each predicate can be transformed to an equivalent form:

$$\Re \quad rel \quad 0$$

where $\Re$ is a real-value function and *rel* is one of $\{\leq, <, =\}$, referred as a *branch function*, which satisfies

1). positive (or zero if *rel* is $<$) when the predicate is false,

2). negative (or zero if *rel* is $=$ or $\leq$) when the predicate is true.

For example, suppose that a program contains the following condition statement

$$if \quad (y \leq 120) \quad \cdots$$

and the *TRUE* branch of the predicate should be taken. Thus, we must find an input that can make the variable y to hold a value smaller than or equal to constant 120 when the condition statement is reached. Let $y_{condition}(x)$ represent the current value of variable y on input x when the program is executed up to the condition statement. Then the branch function $\Re$ can be expressed as follows:

$$\Re(x) = y_{condition}(x) - 120$$

The function is minimal when the *TRUE* branch is taken on the condition statement. So, the problem of dynamic test data generation can be formulated to a function minimization problem [9]. That is, we need to find an input x that can minimize the branch function $\Re(x)$.

Gradient descent is considered as a standard function minimization technique, which performs function minimization by only evaluating the branch function values [6, 10]. In general, gradient descent is faster than global optimization algorithms such as genetic search [9], and is often used in dynamic test data generation, e.g., ADTEST system [10]. We also employ gradient descent to perform function minimization during our *ON-OFF* test point generation for character string predicate border. A shortcoming of using gradient descent technique is that gradient descent algorithms are likely to fail when they meet a local minimum [6, 9]. That is, branch function appears to reach the minimum but it does not. However, our gradient descent algorithm is not subject to this problem (see Section 4.2)

4. Domain testing based on character string predicate

Domain testing has been thought of as a path-oriented testing method. This technique first requests to determine a path that is to be followed. There are a number of path selection strategies reported in the literature [4,11]. Here we focus on how to automatically generate *ON-OFF* test points for character string predicate borders associated with a path, leaving out the account for the test paths selection.

4.1 Character string predicate and predicate slice

A *character string predicate* in programs is of the following form

$$strcmp \, (str_1, str_2) \quad op \quad 0$$

where str_1 and str_2 are character strings or character string variables, and $op \in \{<, =, >\}$. Each character string predicate determines a border.

As described above, dynamic test data generation can be reduced to the problem of function minimization. As a result, we need to construct a branch function with respect to a predicate on a path, and then evaluate the branch function value. Thus, the values of variables in the predicate must be calculated for program inputs. The current values of variables in a predicate can be calculated or collected by using symbolic execution or program instrumentation technique [10]. Unfortunately, symbolic execution encounters some problems in practice [9, 10], and it is impossible to obtain its predicate interpretation for a character string predicate. Program instrumentation technique requires selectively inserting additional codes into the program with appropriate positions so that they can be executed immediately before the predicate is evaluated. That is to say, in the case of an *if* statement, the instrumentation code is injected directly before the statement, but in a *loop* control structure, two instrumentation statements need to be inserted: one immediately before the loop structure and one after the body of statements contained within the control structure. It is costly and time-consuming to design proper instrumentation codes according to various conditions in a program, especially when manual insertion is unavoidable.

In the research reported in this paper, we construct a slice with respect to a predicate to calculate the current values of the variables in the predicate. Suppose that program P is executed along a chosen path π_x on input x. Let *pr* denote a predicate on path π_x, q denote the node

associated with the predicate *pr,* and *y* be a variable in *pr.* Then, a slice is defined by a slicing criterion $C = (x, y^q)$, where y^q represents a variable at position *q* [12,13].

A predicate slice of program *P* based on slicing criterion *C* is a syntactically correct and executable program that is obtained from *P* by deleting zero or more statements and adding the statements that return the values of the variables in *pr.* The predicate slice produces an execution trace π'_x on input *x* for which there exists a corresponding execution position q' such that the value of variable *y* at *q* on π_x is equal to that of *y* at q' on π'_x. In other words, the slice preserves the values of the variables in *pr* on program inputs.

If the position q' could not be reached on input x_1, the slice returns *F*, indicating constraint violation occurrence; otherwise, the value of variable *y* and *T* are returned. For instance, the program *P* shown in Fig.1, which is a variation of the program taken from [14], will be traversed along the path $\pi_x = \{1,2,3,4,5,6,7,10,13,14,15,16,17,19\}$ on input $x\square argc=4$ and *argv[1]="-ceiling", argv[2]="193", argv[3]="A2".* The predicate slice with respect to the predicate *p6* (statement 16) on path π_x, denoted by *predicate_slice_p6,* is presented in Fig.2. In this way, the current values of character string variable *result* and *ceiling* in the predicate *p6* can be obtained by executing the *predicate_slice_p6.* More complete discussions about the predicate slice generation with detailed descriptions can be found in [15]. In this paper, we mainly describe how to generate *ON-OFF* test points in character string domain.

```
int  max(int argc, char ** argv)
1 {  argc--;
2    argv++;
3    if ((argc>0)&&('-'==**argv))
4    {   if (!strcmp(argv[0],"-ceiling"))
5        {    strncpy(ceiling,argv[1],BUFSIZE);
6             argv++; argv++;
7             argc--;   argc--;   }
         else
8        {    fprintf("Illegal option %s.\n",argv[0]);
9             return(0);  }; }
10   if(argc==0)
11   {    fprintf("At least requires one arguments.\n");
12        return(0);  }
13   for(;argc>0;argc--,argv++)
14   {   if(strcmp(argv[0],result)>0);
15        strncpy(result,argv[0],BUFSIZE);  }
16   if (strcmp(result, ceiling)>0)
17       printf("\n max:%s", result);
18   else  printf("\n max:%s", ceiling);
19   return(1); }
```

Fig.1   Program *P*

```
int predicate_slice_p6(int argc, char ** argv, char *
restr1, char * restr2)
1 {  argc--;
2    argv++;
3    if ((argc>0)&&('-'==**argv))
4    {   if (!strcmp(argv[0],"-ceiling"))
5        {    strncpy(ceiling,argv[1],BUFSIZE);
6             argv++; argv++;
7             argc--;   argc--;   }
         else
8    {    return(0);  }; }
10   if(argc==0)
11   {    return(0);  }
13   for(;argc>0;argc--,argv++)
14   {   if(strcmp(argv[0],result)>0);
15        strncpy(result,argv[0],BUFSIZE);  }
16   strcpy(restr1,result);
17   strcpy(restr2,ceiling);
18   return(1);}
```

Fig.2   Predicate_slice_p6

4.2 *ON-OFF* test point generation in character string domain

Now, we describe how to generate *ON-OFF* test point for character string predicate borders by Jeng's simplified domain testing strategy. The problem can be stated as follows:

"Given a character string predicate border associated with a chosen path, the goal is to find a program input pair so that one lies on the given border whereas the other is placed outside this border, and the pair has to be as close as possible."

For this purpose, a problem that we must solve first is how to compare two character strings as well as how to evaluate a branch function $\Re$ with respect to a character string predicate. Moreover, the simplified domain testing strategy requests that *ON* and *OFF* test points are placed as closely as possible; namely the distance of the two points is the shortest. Accordingly, we define a function φ, which maps a character string to a nonnegative integer, satisfying the formula:

$$\varphi(str) = \sum_{i=0}^{L-1} str[i] \times w^{L-i-1} \qquad [1]$$

where *str* is a character string, *L* is its length, w^{L-i-1} is a positive weighting factor representing a weighted value imposed upon each character element of the string, and *w* is equal to 128.

It is easy to see that a character string can be transformed into a unique nonnegative integer by using Eq.1. Thus, the distance between two strings is defined as below:

Definition: Let L_1 and L_2 denote the length of strings str_1 and str_2, respectively. Suppose *L* is the maximum of

L_1 and L_2. Without loss of generality, let $L = L_2$, and $str_1[k] = '\backslash 0', (k < L-1)$. By *the distance between strings* str_1 *and* str_2, represented by $dis(str_1, str_2)$, we mean

$$dis(str_1, str_2) = \left| \sum_{i=0}^{L_1-1} str_1[i] \times w^{L-i-1} - \sum_{i=0}^{L_2-1} str_2[i] \times w^{L-i-1} \right| \quad [2]$$

By *the distance between the* i^{th} *characters of string* str_1 *and* str_2, denoted by $d_i(str_1, str_2)$, we imply

$$d_i(str_1, str_2) = \left| str_1[i] - str_2[i] \right| \quad (0 \le i \le L-1) \quad [3]$$

The distance $dis(str_1, str_2)$ uniquely determines a nonnegative integer.

Consequently, we can construct a branch function $\Re$ with respect to a character string predicate, e.g., $strcmp(str_1, str_2) > 0$, so that its value is positive for an initial input x_0. Namely, let $\Re = dis(str_1 - str_2)$. Since every character element of a string is expressed by its ASCII code (an integer), a practical way is to construct the branch functions for every character. That is to say, we construct $\Re_i$ corresponding to the ith character, i.e., let $\Re_i = \left| str_1[i] - str_2[i] \right|$ so that $\Re_i > 0$, $(i = 0, 1, 2, \cdots, L-1)$. Then, search an adjustment direction for the ith character so that $\Re_i$ can be improved. Each $\Re_i$ can reach its minimum by using gradient descent to perform function minimization. As a result, we obtain two distinct characters that one satisfies $\Re_i \le 0$ whereas the other meets $\Re_i > 0$. The two characters are refined gradually until the distance between them, i.e., $d_i(str_1, str_2)$, becomes the shortest. As each character is determined in turn, the branch function $\Re$ with respect to the predicate can become negative (or zero). Thus, we obtain two points that one satisfies $\Re \le 0$ while the other meets $\Re > 0$. They can be selected as *ON, OFF* test points, respectively, depending on the operator *op*, and the distance between them is the shortest.

In what follows we will explain in detail how to automatically generate *ON-OFF* test points for a character string predicate border by using gradient descent to perform function minimization. The current values of the variables in the predicate are calculated by executing the corresponding predicate slice. Let π be a path in the program under test, *adjustr* represent an adjusted input variable, and *pr* denote a character string predicate on π. Suppose that x_0 is an initial input (selected randomly or by hand) on which the program can be executed to the predicate *pr* along path π.

If string str_1 is equal to str_2 on input x_0, the test generation algorithm does not need to be invoked. We select x_0 and the corresponding input that the variable *adjustr* is added or subtracted by 1 on the last character and the remaining input variables are held constant as *ON*,

OFF test points, respectively, depending on the operator *op*. Otherwise, the corresponding characters of strings str_1 and str_2 are compared from position 0 to *L-1*. That is, a branch function $\Re_i$ is constructed so that $\Re_i = \left| str_1[i] - str_2[i] \right| > 0$ for the ith unequal character. Then, an adjustment direction is searched by modifying the ith character of the variable *adjustr,* denoted by c_i, i.e., let $c_i' = c_i + 1$ or $c_i' = c_i - 1$. If c_i' results in a better $\Re_i$ value than c_i, c_i' replaces c_i, and a proper direction is found; otherwise, if there is another input variable, it is selected as adjusted variable, or else the *ON-OFF* test point generation fails for the predicate border. For instance, suppose that only input variable *instr* is connected with the predicate *pr*, and the predicate slice associated with *pr* implements the function: str_1 ="abc"+*instr*, str_2 =" 2334". In this case, no matter how to adjust the input variable *instr*, str_1 is always greater than str_2. Hence, there is not an *OFF* test point for the border.

When a good direction is found, the adjustment amount is increased (doubled) until either (1) $\Re_i \le 0$, or (2) $\Re_i$ is not improved, or (3) constraint violation occurs, or (4) c_i' is outside of 32 and 127. In the last three cases, we reduce the adjustment amount and the corresponding input is tried again. In the first case of $\Re_i \le 0$, we obtain two distinct characters C_{on} and C_{off} such that C_{on} meets $\Re_i \le 0$ and C_{off} satisfies $\Re_i > 0$. The two characters are refined gradually with the help of another character C_{it} whose initial value is C_{off}. Subsequently, C_{it} is modified by reducing (halving) the adjustment amount. The corresponding input is executed and $\Re_i$ is evaluated. If $\Re_i$ corresponding to C_{it} is negative, then C_{on} takes C_{it} value; otherwise C_{off} takes C_{it} value. The process is repeated until the distance between C_{on} and C_{off}, namely $d_i(str_1, str_2)$, becomes the shortest. If $d_i(str_1, str_2)$ is adjusted to 0, the ith character of the adjusted variable *adjustr* is determined, and the next character, i.e., $i+1\text{©}$ character, is considered. Otherwise, the ith character of the variable *adjustr* takes C_{on}, C_{off} values, respectively, and the corresponding inputs are selected as *ON, OFF* test whereas the algorithm terminates. If the variable *adjustr* ends before $i < L-1$, a space character (ASCII 32) is added before its terminating position. The comparison continues until $i = L-1$. If $\Re_{L-1}$ is adjusted to 0, then $\Re = 0$. The current input can be taken as *ON* (or *OFF*) test point, and the corresponding input where the variable *adjustr* plus or minus 1 on the last character (while other variables keep constant) is selected as *OFF* (or *ON*) test

point, depending on the operator *op*. It is obvious that the distance of the two test points generated in this way is the shortest.

A shortcoming of using gradient descent to perform function minimization is that gradient descent algorithms can fail if a local minimum is encountered. Fortunately, our gradient descent algorithm would not encounter the problem. We note that minimizing a branch function is very difficult if str_1 and str_2 are all involved in an adjusted input variable. In most cases, one of them is not related to the adjusted variable. We assume that str_2 has nothing to do with the adjusted input variable, and c_i represents the ith character of the adjusted variable. Then, at position *i*, we have $\Re_i = |\ str_1[i] - str_2[i]\ |$. In fact, $str_1[i]$ is a function of c_i, denoted as $\vartheta(c_i)$, and $str_2[i]$ is not connected with c_i, which can be thought of as a constant, represented by M. Accordingly, the branch function $\Re_i$ can be expressed as $\Re_i = \vartheta(c_i) - M$ or $\Re_i = M - \vartheta(c_i)$.

It is easy to see that $\Re_i$ is a monotonically increasing or decreasing function, i.e., the adjustment for each character is not restricted to a localized region of $\Re_i$. $\Re_i$ can reach its minimum so that each character of the adjusted variable is determined in turn. As a result, the function minimization of $\Re$ does not suffer from the local minimum problem.

4.3 Automatic *ON-OFF* test point generator

Here, we present our automatic *ON-OFF* test point generator for programs written in C programming language, which is developed on the basis of the idea described above.
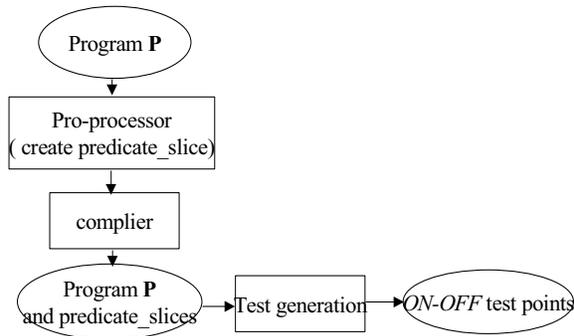
Fig.3 The architecture of automatic test generator

As shown in Fig.3, the program under test is first processed by a *pre-processor*, which completes the construction of predicate slices with respect to every predicate on paths. The resulting predicate slices are complied, and the executable code is gained. Then, the generation algorithm is employed to generate *ON-OFF* test

points for a character string predicate border associated with a program path.

5. A case study

We use an example to illustrate *ON-OFF* test point generation for a character string predicate border associated with a program path.

Suppose program *P* in Fig.1 is traversed along path $\pi_x = \{1,2,3,4,5,6,7,10,13,14,15,16,17,19\}$ on input $x\ \square$ *argc=4*, *argv[1]="-ceiling"*, *argv[2]="193"* and *argv[3]="A2"*. The predicate *P6: strcmp(result, ceiling)>0)* (statement 16) refers to a given character string predicate, and *argv[3]* is an adjusted input variable. The predicate slice with respect to *x* and π_x, *predicate_slice_p6*, is produced by the pro-processor. By executing the predicate slice, we obtain the current value of variables *result* and *ceiling* in the predicate *p6* on input *x*, that is, *result* = "A2", *ceiling* = "193".

According to the test generation algorithm, the *ON-OFF* test points for the predicate border are

ON test point $\square$ *argc=4*, *argv[1]="-ceiling"*, *argv[2]="193", argv[3]="192"*.

OFF test point $\square$ *argc=4*, *argv[1]="-ceiling"*, *argv[2]="193", argv[3]="193"*.

It is clear that the distance between *ON* and *OFF* test point is shortest, only equaling to 1. The details are described as below:

At position 0, *argv*[3][0]= 'A'. Here, *result*[0]= 'A', ASCII is 65, and *ceiling*[0]= '1', ASCII is 49. So, $\Re_0 = result[0] - ceiling[0] = 16$, and ' - ' is taken as adjustment direction. The determination of the 0th distinct character is demonstrated in table 1.

SETP	C_0	C_0'	$\Re_0$	*Ceiling[0]*	*dir*
1	65	64	15		
2	64	62	13		
4	62	58	9	49	-
8	58	50	1		
16	50	34	<0		

Table1(a) Search distinct characters at position 0

SETP	C_{off}	C_{on}	C_{it}	$\Re_0$
16	50	34	50-8=42	<0
8		42	50-4=46	<0
4		46	50-2=48	<0
2		48	50-1=49	=0

Table1(b) Refine distinct characters at position 0

When $\Re_0 < 0$ we get two distinct characters whose ASCII are 50 and 34, respectively. The two distinct characters are refined gradually until $\Re_0 = 0$. Here,

COMPUTER SOCIETY

C_{it} =49. Let *argv*[3][0]= 49*,* then the 0th character of input variable *argv*[3] is determined. Now *argv*[3] ="11".

Then, we compare the 1th character of variable *result* and *ceiling*. At position 1, *argv*[3][1]= '2'. Here, *result*[1]='2'*,* ASCII is 50, and *ceiling*[1]='9', ASCII is 57. So $\Re_1 = ceiling[1] - result[1] = 7$, and '+' is regarded as adjustment direction. The determination of the 1th distinct character is shown in table 2.

SETP	C_1	C_1'	$\Re_1$	*Ceiling[1]*	*dir*
1	50	51	6		
2	51	53	4	57	+
4	53	57	0		

Table2 Search distinct characters at position 1

When C_1' =57, we derive $\Re_1 = 0$. Thus, it is of no use for the process of refining distinct character. Let *argv*[3][1]=57, then the 1th character of input variable *argv*[3] is determined. Here *argv*[3]="19". Continue to compare the next character.

At position 2, *argv*[3][2]= '\0', but *ceiling*[2]='3'. Let *argv*[3][2]=' ', *argv*[3][3]='\0'. So, *result*[2]= ' ', and '+' is treated as adjustment direction. The process of determining the 2th character of input variable *argv*[3] is similar to that of the 0th character.

When C_{it}=51, $\Re_2 = 0$. Let *argv*[3][2]=51, then *argv*[3]="193". The algorithm terminates since *ceiling*[3]='\0'. According to the operator of the predicate *p6*, the current input is selected as *OFF* test point; namely, *OFF* test point is *argc=4, argv[1]="-ceiling", argv[2]="193", argv[3]="193"*. The corresponding input that the adjusted variable *argv*[3] minus 1 on the last character, i.e., *argv[3][2]=argv[3][2]*-1, while other input variables keep constant, is taken as *ON* test point. So, *ON* test point is *argc=4, argv[1]="-ceiling", argv[2]="193", and argv[3]="192"*. It can be seen that the distance of *ON-OFF* test points obtained in this way is the shortest.

6. Conclusion

The objective of domain testing is to detect domain errors in programs. Nevertheless, all recent domain testing strategies have been limited to programs in which character string predicates are not taken into consideration. The same weakness is found in many currently available test data generation system. In this paper, we present a novel approach to automatically generate *ON-OFF* test points for character string predicate borders associated with program paths, and develop a corresponding test data generator by Jeng's simplified domain testing strategy.

Symbolic execution or program instrumentation is not involved in the system. Instead, a predicate slice is constructed to calculate the current values of variables in the predicate, avoiding the problems found in symbolic execution and the cost of designing proper instrumentation

codes.

To our knowledge, this is the first approach to automatic test data generation based on character string predicates. The preliminary experimental results show that the methodology is promising and effective.

Reference

[1] M. R. Lyu, S. Rangarajan, and A.P.A. van Moorsel, "Optimal Allocation of Test Resources for Software Reliability Growth Modeling in Software Development," *IEEE Transactions on Reliability,* Vol. 51, No. 2, June 2002, pp. 183-192.
[2] P. C. Jorgensen. "Software Testing: A Craftsman's Approach". CRC Press LLC. 2002.
[3] L. J. White and E. I. Cohen, "A Domain Strategy for Computer Program Testing," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 3, May 1980, pp. 247-257.
[4] S. J. Zeil and F. H. Afifi and L. J. White. "Detection of Linear Errors via Domain Testing." *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 4, October 1992, pp. 422-451.
[5] B. Jeng and E. J. Weyuker, "A Simplified Domain-Testing Strategy." *ACM Trans. Software Engineering and Methodology,* Vol.3, No.3, July 1994, pp. 254-270.
[6] A. Hajnal and I. F. orgacs. "An Applicable Test Data Generation Algorithm for Domain Errors," *ISSTA'98, Proceedings of ACM SIGSOFT International symposium on Software Testing and Analysis, Florida*, USA, March 2-5, 1998, pp. 63-72.
[7] W. E. Howden. "Reliability of the Path Analysis Testing Strategy." *IEEE Transactions on Software Engineering,* SE-2, 3, 1976, pp. 208-215.
[8] B. Korel. "Automated Software Test Data Generation", *IEEE Transactions on Software Engineering,* Vol.16, No.8, Auguest.1990, pp. 870-879.
[9] C. C. Michael, G. McGraw, and M. A. Schatz. "Generating Software Test Data by Evolution," *IEEE Transactions on Software Engineering*, Vol.27, No.12, Dec. 2001, pp. 1085-1110.
[10] M. J. Gallagher and V. L. Narasimhan. "Adtest: A Test Data Generation Suite for Ada Software System." *IEEE Transactions on Software Engineering,* Vol. 23, No. 8, Aug. 1997, pp.473-484.
[11] L. S. Koh, M.T. Liu. "Test Path Selection Based on Effective Domains," *Proceedings of International Conference on Network Protocols*, 1994, pp. 64 –71.
[12] A. Beszedes, T. Gergely, Z. M. Szabo, J. Csirik and T. Gyimothy. "Dynamic Slicing Method for Maintenance of Large C Program," *Fifth European Conference on Software Maintenance and Reengineering,* 2001, pp. 105-113.
[13] F. Tip. "A Survey of Program Slicing Techniques", *Journal of Programming Languages*, Sept.1995, 3(3), pp. 121-189.
[14] B. Marick. "The Craft of Software Testing," *PTR Prentice Hall*, NJ, 1995.
[15] R. Zhao, "Research on Software Testing Methodologies", *Ph.D. thesis*, Chinese Academy of Science, 2001.