SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2010; **20**:209–236
Published online 6 July 2009 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.414

Automatic string test data generation for detecting domain errors

Ruilian Zhao[1,*,†], Michael R. Lyu[2] and Yinghua Min[3]

[1]*Department of Computer Science, Beijing University of Chemical Technology, Beijing 100029, China*
[2]*Department of Computer Science, Chinese University of Hong Kong, Hong Kong*
[3]*Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080, China*

SUMMARY

Domain testing is designed to detect domain errors that result from a small boundary shift in a path domain. Although many researchers have studied domain testing, automatic domain test data generation for string predicates has seldom been explored. This paper presents a novel approach for the automatic generation of *ON–OFF* test points for string predicate borders, and describes a corresponding test data generator. Our empirical work is conducted on a set of programs with string predicates, where extensive trials have been done for each string predicate, and the results are analysed using the SPSS tool. Conclusions are drawn that: (i) the approach is promising and effective; (ii) there is a strong linear relationship between the performance of the test generator and the length of target string in the predicate tested; and (iii) initial inputs, no shorter than the target string and with characters generated randomly, may enhance the performance in the test data generation for string predicates. Copyright © 2009 John Wiley & Sons, Ltd.

Received 16 August 2007; Revised 14 April 2009; Accepted 28 April 2009

KEY WORDS: domain testing; string predicate; dynamic test data generation; *ON–OFF* test point

*Correspondence to: Ruilian Zhao, Department of Computer Science, Beijing University of Chemical Technology, Beijing 100029, China.
†E-mail: rlzhao@mail.buct.edu.cn

Contract/grant sponsor: National Natural Science Foundation of China; contract/grant number: 60473032
Contract/grant sponsor: Beijing Natural Science Foundation; contract/grant number: 4072021
Contract/grant sponsor: Hong Kong Research Grants Council; contract/grant number: CUHK4150/07E

1. INTRODUCTION

Software testing plays an important role in the process of guaranteeing software quality and reliability [1]. One of the most difficult and expensive problems in software testing is the effective generation of test data. Test data generation is the process of creating program inputs that satisfy some testing criterion [2]. Obviously, manually developing a large test data set to satisfy a testing criterion is usually expensive, laborious, difficult and error-prone. If test data could be automatically generated, the cost of software testing would be significantly reduced.

It is usually observed that the input data near the boundary of a domain are more sensitive to program faults and should be carefully checked. A domain testing strategy is very effective in verifying the correctness of the boundary of a path domain; however, such a domain strategy is hard to implement since the strategy requires test data generated on and near the boundary, and the test generation is more difficult when some of the constraints are nonlinear or in a discrete space [3]. Domain testing has received a certain amount of research attention [3–9], although automatic *ON–OFF* test generation has mostly focused on numeric data types and ignored string data. That is to say, most recent domain testing strategies have been limited to programs in which each predicate can contain Boolean data, numeric data, relational operators, or binary Boolean operators, etc. but string data have not been taken into account. The few studies that have addressed string test generation have concentrated on string mutation operations such as deletion, insertion and substitution rather than domain testing [10]. This negligence seriously restricts the usefulness of the domain testing strategy in practice, where string predicates are widely used in modern programming techniques.

In the research reported in this paper, we define the distance between two strings and develop a test data generator to automatically generate *ON–OFF* test points for string predicate borders during unit testing. In order to identify how close the current input is near the border, an objective function is associated with the string predicate, and a greedy heuristic search algorithm is used to guide the search for an input string pair, namely *ON–OFF* test points, such that the *ON* test point lies on the given string predicate border, whereas the *OFF* test point is selected just outside the border, and is as close to the *ON* test point as possible. The current values of variables in the predicate are calculated or collected by the program instrumentation techniques. Each character element of the target string in the string predicate under test is determined in turn by the search technique so that the *ON–OFF* test points corresponding to the predicate are automatically generated.

In order to investigate the effectiveness of the string *ON–OFF* test generation approach, a number of experiments have been conducted on empirical programs containing string predicates; these programs are derived from books or Web sources. Four hundred and eighty trials have been done for each string predicate, and the experimental outcomes are analysed in detail using the statistical analysis tool such as SPSS. The results presented in Section 5 show the effectiveness of this methodology.

The remainder of this paper is organized as follows. Section 2 introduces the basic terminology and domain testing strategies. Section 3 briefly reviews the dynamic test data generation technique. Section 4 describes the main principle of the automatic *ON–OFF* test point generation with respect to string predicate borders, and gives an example to illustrate in detail how the test generation approach works. Section 5 reports the empirical analysis of the string test data generation. Finally, conclusions are given in Section 6.

DOI: 10.1002/stvr

2. BASIC TERMINOLOGY AND DOMAIN TESTING

2.1. Basic terminology

A program structure can be represented by a *control flow graph*, denoted by $G=(V, E, s, e)$, where V is a set of nodes of G, E is a set of edges and s and e $(s, e \in V)$ are, respectively, the unique entry and exit nodes. Here, nodes stand for statements, and edges indicate the possible flow of control between statements. A *subpath* π from v_i to v_k is a sequence of nodes $\pi = < v_i, v_{i+1}, \ldots, v_k >$, where each adjacent pair (v_{i+j}, v_{i+j+1}) is an edge in G for $0 \leq j < k-i$. A *prefix path* is a subpath that starts from the entry node s, and a (*complete*) *path* is a subpath from the entry node s to the exit node e. An edge (v_i, v_j) is called a *branch* if v_i corresponds to a decision statement at which the control flow has two or more alternative execution routes, such as **if–then–else**, **switch**, **for** or **while** statements in C programs. Each branch in a control flow graph can be labeled with a *predicate* that describes the conditions under which the branch will be traversed. A predicate is usually connected with a *predicate interpretation* that is obtained by replacing each variable appearing in the predicate with its symbolic value in terms of input variables. Each path is associated with a *path condition* that is the conjunction of all the predicate interpretations that are taken along the path. The path condition represents the constraints that have to be satisfied for inputs in order to execute the path. These inputs construct the *path domain*. The boundary of a path domain is determined by the predicates encountered along the path. Each predicate corresponds to a segment of the boundary called a *border*. The border to be tested is called *the given border*, which may or may not be correctly implemented. The one that corresponds to the given border in a correct program is called the *correct border*. When a given border differs from the correct border, *a border shift* is said to occur [4,7].

2.2. Domain testing strategy

The critical feature of a good testing strategy is its ability to guide test case selection to maximize the possibility of finding hidden faults. Thus, some analysis of how faults occur in a program is helpful and necessary for choosing a test strategy. As identified by Howden, program errors[‡] can be classified into three categories: computation errors, missing-path errors and domain errors [11], where *error* is used to represent the difference between the actual and the expected output. A program is said to contain 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 *missing-path* error appears when some path that should be traversed by a specific input for the program is missing. A *domain error*, which can be manifested by one of the given borders being shifted from its correct location, occurs when a specific input traverses a wrong path because of faults existing near the boundary of a path domain.

Detecting a domain error may be thought of as determining whether a border shift has occurred. A domain testing strategy, first proposed by White and Cohen, is used to detect these types of errors.

[‡]In the IEEE Standard Glossary of Software Engineering Terminology, the two words ‘error’ and ‘fault’ have different meanings, and ‘error’ used here should be replaced with ‘fault’. But, for historical reasons, the terms ‘computation error, missing-path error and domain error’ are still used in this field today. Thus, the words ‘error’ and ‘fault’ are used interchangeably in this paper.

DOI: 10.1002/stvr

For a linear predicate with a total of n distinct numeric variables, the strategy involves designing n *ON* test points and one *OFF* test point. The *ON* test points lie on the given border, while the *OFF* test point is selected just outside the border, and is close to these *ON* test points [4]. However, Clarke *et al.* discovered that some domain errors went undetected by White and Cohen's strategy, and presented two alternatives $V \times 1$ and $V \times V$, assuming that the border holds V vertices. Their strategies select *V ON* test points with one for each vertex and one or *V OFF* test points, respectively, to be placed at a uniform distance away from the border [5]. Zeil *et al.* extended domain testing to detect linear errors in a nonlinear predicate [6]. Later, Jeng and Weyuker developed a simplified domain testing strategy, which generates one *ON* and one *OFF* test points in any dimension for an inequality border corresponding to a predicate that contains one of the operators $\leq$, $<$, $\geq$ or $>$. For an equality or non-equality border associated with the operator $=$ or $\neq$, one *ON* test point and two *OFF* test points are requested. The strategy requires that the *ON* test point should be selected on the border, whereas the *OFF* test point should be placed outside the border, and the *ON–OFF* point pairs have to be as close to each other as possible. In this case, the only way a border shift can escape detection is the correct border passing through in between the *ON* and *OFF* points. Since the *ON–OFF* point pairs are very close to each other, this is unlikely to happen [7]. Hajnal and Forgacs introduced an algorithm to generate *ON–OFF* test points according to the simplified domain testing strategy with some manual assistance [8]. In addition, Jeng integrated domain testing and data flow testing to enhance their effectiveness in fault detection [9]. The majority of the systems developed by applying these techniques have focused on generating numeric *ON–OFF* test points, though Jeng and Weyuker made a suggestion on how strings may be handled at the end of their paper [7]. In this paper, however, we focus on character string domain testing.

3. DYNAMIC TEST DATA GENERATION

3.1. Dynamic test data generation

There are many automatic test data generation approaches. The most often used are random test data generation, symbolic execution-based test data generation and dynamic test data generation [12].

Random test data generation develops test data at random until an appropriate input is found [12,13]. This approach is easy to understand and commonly employed in the literature. However, randomly generated test data have difficulties in revealing domain errors since the *ON–OFF* test points with respect to a given border can be very specific in the wide input domain, and the likelihood of finding suitable test points randomly can be extremely low. For example, consider generating an *ON* test point to check the predicate in the program fragment shown below.

```
Procedure(x: string)
{ ...
 strncpy(v,x,5);                  /* copy initial 5 characters of x to v */
 strupr(v);                       /* convert each lowercase character to uppercase */
 if (strcmp(v, "LEFT")<0) ...;    /* compare v and 'LEFT' lexicographically */
 ...
}
```


DOI: 10.1002/stvr

where x is a string input variable and v is a string variable. The possibility for a randomly generated input to set the variable v to be equal to the literal *LEFT* is very low. Thus, in practice, random test data generation performs poorly, and is most often used merely as a benchmark for the evaluation of guided search methods.

The basic idea in a symbolic execution system is to allow numeric variables to take on symbolic values [14,15]. However, symbolic execution is computationally very intensive and a number of technical problems are often encountered in practice, such as indefinite loops, subprogram calls and array references [12], especially for programs containing string predicates. For instance, considering the above codes fragment, in general, the relationship between the string v and the input x may not be represented in a straightforward way. As a result, it is difficult to express the value of variable v in terms of the symbolic value of the input x in the predicate that follows.

Dynamic test data generation is the most commonly used approach for developing test data. During dynamic test data generation, if some desired test requirement is not reached, data generated in each test execution are used to identify how close the test input is in meeting the requirement. With the aid of feedback, test inputs are gradually modified until one of them satisfies the requirement. For example, suppose that a program contains the condition statement

$$\text{if } (y \leq 128)\ldots$$

where y is a variable. The *True* branch of the predicate should be taken when the program is executed up to the condition statement. Thus, we must find an input that can make the variable y hold a value smaller than or equal to the constant 128 when the condition statement is reached.

Without loss of generality, a predicate is assumed to be of the form $E_1\ op\ E_2$, where E_1 and E_2 are the arithmetic expressions and op is a relational operator. As discussed in References [16,17], each predicate $E_1\ op\ E_2$ can be transformed to an equivalent form as

$$\Re \quad op_1 \quad 0$$

where $\Re$ is a real-valued function, referred to as an *objective function*, and $\Re$ and op_1 are given in Table I. The function is either (1) positive (or zero if op_1 is '$<$') when the predicate for the required branch is false or (2) negative (or zero if op_1 is '$=$' or '$\leq$') when the predicate is true.

A simple way to calculate the current value of variable y in the predicate is to execute the program up to the condition statement and record the value of y. Let $y_{condition}(x)$ represent the current value of variable y on input x when the program is executed up to the predicate. Then the objective function can be expressed as follows:

$$\Re(x) = y_{condition}(x) - 128$$

Table I. The objective functions for predicates.

Predicate	Objective function $\Re$	op_1
$E_1 > E_2$	$E_2 - E_1$	$<$
$E_1 \geq E_2$	$E_2 - E_1$	$\leq$
$E_1 < E_2$	$E_1 - E_2$	$<$
$E_1 \leq E_2$	$E_1 - E_2$	$\leq$
$E_1 = E_2$	$abs(E_1 - E_2)$	$=$
$E_1 \neq E_2$	$-abs(E_1 - E_2)$	$<$

DOI: 10.1002/stvr

The function $\Re(x)$ is negative or zero when the *True* branch is taken to reach the conditional statement. Therefore, the objective function $\Re(x)$ can be used to guide the search for test data. Furthermore, it also indicates how close the current input is to meeting the predicate requirement; in other words, the value of $\Re(x)$ determines how many modifications must be performed to transform an initial input into a solution, for instance, an *ON* test point.

Various dynamic test data generation methods have been used, including a greedy heuristic search algorithm, a genetic algorithm, a simulated annealing algorithm and so on [16–19]. For example, Harman *et al.* apply testability transformation to improve the performance of evolutionary test data generation in the presence of a flag variable [18]. Mansour and Salame compare a genetic algorithm and a simulated annealing algorithm with each other and with a known greedy heuristic search algorithm, for integer- and real-valued programs. Their work shows that a genetic algorithm is faster than a simulated annealing algorithm; however, a greedy heuristic search algorithm, when it succeeds in finding test data, is the fastest [19]. Therefore, in our research, we employ a greedy heuristic search algorithm to derive *ON–OFF* test points for string predicate borders associated with a path domain.

3.2. Greedy heuristic search algorithm

In this section, we describe how a greedy heuristic search algorithm works[§]. Suppose x_0 is an original input on which the program is executed up to a predicate, and the *False* branch of the predicate is taken. An objective function can be constructed, whose value is positive for input x_0. In order to search for a good adjustment direction, a new input x' is created via a small increment or decrement with respect to x_0 on an input variable that has influence on the predicate, while keeping all other input variables unchanged. The program is executed on input x' and the objective function is evaluated. If neither an increase nor a decrease of the input variable causes an improvement, i.e. a decrement of the objective function, another input variable is selected.

When an appropriate direction is found, i.e. the program execution reaches the predicate and the objective function is improved, a larger adjustment is made in this direction. Then, the program is executed on the new input, and the objective function is evaluated again. If the program no longer reaches the predicate, i.e. a constraint violation occurs, an adjustment continues in this direction with a smaller amount. If the objective function is not further decreased, the last value of the objective function is retained, and a new direction is searched on the previous input. If the positive minimum of the objective function is located, an adjustment direction is searched from this minimum using another input variable. The cycle repeats either until the objective function becomes negative or zero, meaning the input that will cause execution of the desired branch of the predicate has been found, or until no further decrease can be made for any input variable, meaning there is no input that can make the *True* branch of the predicate to be taken.

A greedy heuristic search algorithm falls into the category of a local search technique. A shortcoming is that the algorithms are likely to fail when they meet a local minimum. This occurs when the objective function appears to have reached the global minimum, but in fact it has not. However, our algorithm is not subject to this problem (see Section 4.5).

[§] The framework of the algorithm is originated from Korel's dynamic approach which is aimed at deriving test data to cover the selected paths.

DOI: 10.1002/stvr

4. DOMAIN TESTING BASED ON STRING PREDICATES

Domain testing has been thought of as a path-oriented testing method. This technique first determines a program execution path to be followed. A number of path selection strategies have already been reported in the literature [20,21]. In this paper, we focus on how to automatically generate *ON–OFF* test points for string predicate borders associated with a path.

4.1. Character string search space

Although modern software uses 16-bit character strings, an empirical investigation of the character distribution shows that the typical strings contain only characters in the ASCII range 32–126, while 127 is non-printable. This means that the 16-bit character set does not need to be fully examined when generating test data for practical programs. Let us consider some programs. The empirical programs in Table II are open source C/C++ programs downloaded from Web sources. The size of each program is given in terms of lines of code (*LOC*), which is the sum of data declaration and executable statements. *NOAC* is the number of all characters in each program. *NOPC* is the number of printable characters within the range from the space character to the '∼' character, i.e. ASCII values between 32 and 126. *NA32-127* is the number of the characters from ASCII value 32–127. *Printable%*, *A32-127%*, *A0-127%* and *A0-255%*, respectively, are the percent of characters with ASCII values in 32–126, in 32–127, in 0–127 and in 0–255 as a proportion of all the characters in the source codes. In these programs, the shortest includes 122 *LOC* and the longest includes 6 684 201 *LOC*. For each program, *A0-127%* is the same as *A0-255%*, averaging 99.99%, and the number of printable characters is equal to the number of characters between 32 and 127 except for the program *linux-2.6.17-rc1*, which contains only 6 ASCII 127 characters in over 18 million characters. *A0-127%* is higher than *A32-127%* since *tab*, *LF* and *CR* characters, with ASCII 9, 10 and 13, respectively, are counted in *A0-127%*, but not in *A32-127%*. We therefore assume that it is very unlikely that non-printable characters occur in program string input, and hence characters outside the ASCII range 32–126 are excluded entirely in our test data generation.

4.2. String predicate and string distance

A *string predicate* in programs is a string ordinal predicate that consists of at least one string variable or one string comparison function, e.g. *strcmp*() in C language. As for numerical predicates, a string predicate can be simple or compound. A *simple string predicate* is of the following form:

$$str\_cmp(str_1, str_2)\ op\ \ 0$$

where *str_cmp* is a function that compares two strings, such as *strncmp*( ), str_1 and str_2 are string literals or variables, and operator *op* is a member of $\{<, \leq, =, \neq, \geq, >\}$. A *compound string predicate* is a Boolean combination (*NOT*, *AND, OR*) of one, two or more simple string predicates. Each string predicate determines a border, called a string predicate border.

In dynamic test generation, the test input is successively modified in order to bring it ever closer to satisfying a current test requirement by evaluating an objective function, which heuristically tells how close each input has come to meeting the requirement. This process is equivalent to conducting a minimization of the objective function. Thus, the problem that must be solved first is to find how

DOI: 10.1002/stvr

Table II. The result of empirical investigation for C/C++ programs.

File name	LOC	NOAC	NOPC	NA32-127	Printable (%)	A32-127 (%)	A0-127 (%)	A0-255 (%)
backdoor	122	2583	2461	2461	95.276809	95.276809	100	100
chmod	192	5706	5514	5514	96.63512	96.63512	100	100
des	716	21941	19926	19926	90.81628	90.81628	100	100
exe2bin(dos)	303	8667	8364	8364	96.50398	96.50398	100	100
gcc	856137	53984215	51460877	51460877	95.325785	95.325785	99.999887	99.999887
gif2pcx	292	5495	5203	5203	94.686078	94.686078	100	100
grep(tc)	676	16852	16176	16176	95.988606	95.988606	100	100
LHA	2026	51677	44453	44453	86.02086	86.02086	97.563713	97.563713
linux-2.6.17-rc1	6684201	195375505	181453529	181453535	92.874246	92.87425	99.999487	99.999487
MD5C	662	18085	17264	17264	95.460326	95.460326	99.579762	99.579762
mem(dos)	189	4727	4538	4538	96.001692	96.001692	100	100
move	493	13367	12866	12866	96.251963	96.251963	100	100
unzip	953	19714	18760	18760	95.160799	95.160799	99.994927	99.994927
Total	8546962	249528534	233069931	233069937	93.404119	93.404122	99.999038	99.999038

DOI: 10.1002/stvr

to compare two strings, as well as how to evaluate an objective function with respect to a string predicate. Some known string comparison metrics, such as the Hamming distance [22] and the edit distance [10], can be found in the literature. These string distances attempt to measure the degree of dissimilarity between two strings.

4.2.1. Hamming distance

Hamming distance was originally conceived for detection and correction of errors in digital communication. It assesses the difference between two messages, which is expressed by the number of characters that need to be changed to obtain one from another. The Hamming distance was defined as the number of places in which the two strings differ, i.e. have different characters. e.g. 0101 and 0110 have a Hamming distance of two, whereas 'Butter' and 'ladder' are 4 characters apart. However, the definition takes no account of character set ordering.

4.2.2. Edit distance

The edit distance (or Levenshtein distance) is derived from the consideration of three operators that perform character insertion, deletion and substitution. The edit distance defines the smallest number of insertions, deletions and substitutions, required to change one string into another. Adding or removing a character is known as an *edit operation*. E.g. 'Butter' and 'ladder' have an edit distance of 4 because four substitutions are sufficient to match these two strings.

Alshraideh and Bottaci [10] use an edit distance as the cost function to guide the test generation in achieving branch coverage by a genetic algorithm for string data, and they have a bias to use program literals from the program under test as initial candidates in the test generation. However, if there is a comparison statement such as *if* ($str_1 <= $'*HELLO*)' in a program, in general, variable str_1 is not an input variable. The input may be processed by any number of statements before the string comparison is made, and in a great number of instances, it may not be possible to determine how a program transforms its input. In addition, there exist string predicates that do not contain a string literal. Moreover, even if, for example, *XHELLO* is a candidate solution, in general, deleting *X* from *XHELLO* does not guarantee str_1 equal to *HELLO*. Furthermore, the edit distance fails to consider the ordering of character set. For instance, a comparison between the three strings 'abc', 'def' and 'xyz' yields the same Hamming distance and edit distance, namely 3. This is intuitively unreasonable as two strings are, in general, compared lexicographically in programming language, such as string comparison in C library functions, where the distance between 'abc' and 'def' is obviously smaller than that between 'def' and 'xyz'.

Especially in domain testing strategy where the *ON–OFF* test point pair has to be as close to each other as possible, Hamming distance and edit distance are clearly unsuitable to be an objective function to guide the search for test data. Consider a particular situation that str_1 in string predicate ($str_1 <= $'*HELLO*') is an input variable and the *ON* test point is $str_1 =$ '*HELLO*'. What is the *OFF* test point? There are a lot of conceivable strings causing Hamming distance or the edit distance equal to the lowest value, namely 1, such as *XHELLO*, *HLLO*, *KELLO*, etc. But, only string *HELLP* is appropriate to be used as an *OFF* point because it is the only string just greater than *HELLO* by 1 in string comparison.

DOI: 10.1002/stvr

4.2.3. String ordinal distance

In order to overcome the above problem, we define a string ordinal distance. First, a function φ is defined to map a string to a non-negative integer as follows:

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

where *str* is a string, L is its length, $str[i]$ is the ASCII value of the ith character of string *str*, w^{L-i-1} is a positive weighting factor representing a weighting value imposed on each character element of the string and w is set to 127. A character string is then mapped to a non-negative integer based on units of 127.

Property. *Suppose S is a set of strings, and N_+ is a set of non-negative integers. Let $\varphi(str)$ be defined as in Equation* [1], *and map S to N. Then $\varphi(str)$ is a one-to-one function from S* to N_+.

The property has been proved in [23]¶. It is easy to see that a string can be transformed into a unique non-negative integer by using Equation [1]. It is noted that $\varphi(str)$ may be very large for long strings, which are too large to be represented using language provided integer data types. However, we are not going to really calculate $\varphi(str)$ in our algorithm, but to compare and search for the nearest strings.

Definition. Let L_1 and L_2 denote the length of strings str_1 and str_2, respectively. Suppose $L = \max(L_1, L_2)$, where $\max(L_1, L_2)$ is the maximum of L_1 and L_2. Without loss of generality, let $L = L_2$ and $str_1[k] = '\backslash 0'$, $(k = L_1 \leq L)$; in other words, the strings are left aligned and absent characters are treated as nulls. By *the ordinal distance between string str_1 and str_2*, represented by $Dis(str_1, str_2)$, we mean

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

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

$$d_i(str_1, str_2) = \begin{cases} |str_1[i] - str_2[i]|, & 0 \leq i \leq L_1 - 1 \\ str_2[i], & L_1 \leq i \leq L - 1 \end{cases} \tag{3}$$

where the comparison between str_1 and str_2 (or str_1 [i] and str_2[i]) is related to the lexicographic order.

¶In our early research [23], we discussed how to generate test data for a given path including character string predicates. The test input string was generated automatically for program paths by using the greedy heuristic search algorithm and the investigation on test generation for each feasible path of a Max program illustrates that the search algorithm is more economical than the gradual descent and the random algorithm under the same coverage. In this paper, we consider how to generate ON–OFF test points in a string domain by applying the greedy heuristic search algorithm, and further correct the distance definitions between two strings as well as between ith characters. Moreover, a number of experiments have been conducted on empirical programs containing string predicates in order to evaluate the effectiveness of the string ON–OFF test generation.

DOI: 10.1002/stvr

The distance $Dis(str_1, str_2)$ uniquely determines a non-negative integer. For example, if $str_1 =$ '*abcd*', $str_2 =$ '*ab-xy*', then $L=5$ and $Dis(str_1, str_2) = |\varphi(str_1) - \varphi(str_2)| = [' a' * 127^4 + ' b' * 127^3 + ' -' * 127^2 + ' x' * 127 + ' y'] - [' a' * 127^4 + ' b' * 127^3 + ' c' * 127^2 + ' d' * 127] = (99 - 95) * 16\,129 + (100 - 120) * 127 + (0 - 121) = 61\,855$.

In addition, $Dis($'*abc*',$'$*def*$')=\varphi(def) - \varphi(abc) = 48771$, and $Dis($'*def*',$'$*xyz*$')=\varphi(xyz) - \varphi(def) = 325\,140$.

4.3. ON–OFF test point generation in a string domain

In this section, we describe how to generate *ON–OFF* test points with respect to a string predicate border based on the simplified domain testing strategy. The problem can be stated as follows:

'Given a string predicate border, 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 together as possible.'

For this purpose, we develop an *ON–OFF* test point generator for the programs written in C Language. As shown in Figure 1, the program under test is first inserted with additional codes manually before each decision statement, so that the current values of variables in the predicates can be calculated or collected. The instrumented program is then compiled, and executes as a sub-program of the test generator. According to each string predicate, the test generator constructs an

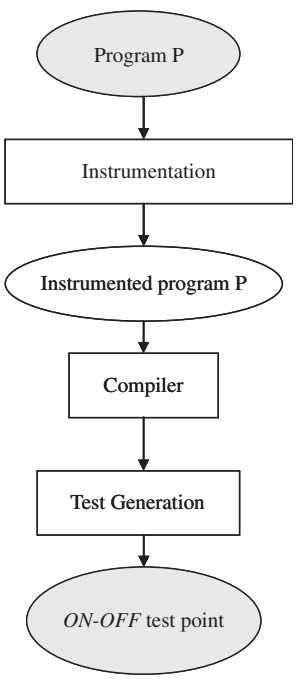

Figure 1. The framework of the *ON–OFF* test generator.

DOI: 10.1002/stvr

objective function $\Re$ and $\Re$ is evaluated once a new input is generated. The process is repeated until $\Re$ becomes zero (or negative) or until no further decrease can be made. In the first case, two input strings are obtained, of which one satisfies $\Re \leq 0$ while the other meets $\Re > 0$. The test generator invokes the function $dis\_min()$ to minimize the distance between the two strings. The two corresponding inputs are selected as *ON* and *OFF* test points, respectively. In the second case, the other input variables are taken into account.

It is observed that many string predicates consist of string comparison functions such as *strcmp*( ) or *strncmp*( ) in C language. Therefore, standard string comparison functions from ANSI C are considered in our test point generator.

In what follows we will explain in detail how to automatically generate *ON–OFF* test points for a string predicate border[‖]. Suppose that x_0 is an initial input (selected randomly or by hand) on which the program can be executed to a string predicate, e.g. $strcmp(str_1, str_2) > 0$, along a given path, and the path condition cannot be satisfied for the string predicate. First, we construct an objective function $\Re$ with respect to the string predicate, whose value is positive for the initial input x_0. Specifically, let $\Re = Dis(str_1, str_2) > 0$. Since $\varphi(str)$ may be very large for long strings, $\Re$ may be very large. In order to make $\Re < 0$, a great deal of $\Re$ evaluation may be required. On the other hand, if a string is considered one character at a time, and each character after plus or minus an adjusted amount, is still included in 32–126, then the number of evaluating $\sum \Re_i$ in the ON-OFF test generation would be much lower than that of $\Re$. Thus, a practical approach is to construct an objective function for each character of string str_1 and str_2. That is to say, we can construct $\Re_i$ corresponding to their ith characters, i.e. let $\Re_i = d_i(str_1, str_2) > 0$, $(i = 0, 1, 2, \ldots, L-1)$ and $L = \max(L_1, L_2)$. Second, an appropriate direction on the ith character of the current adjusted input variable, named *adjustr*, is sought so that $\Re_i$ can be improved (decreased). Each $\Re_i$ can arrive at a negative or zero value by using a greedy heuristic search algorithm. As a result, we obtain two distinct characters such that one satisfies $\Re_i \leq 0$, whereas the other meets $\Re_i > 0$. The two characters are refined gradually until the distance $d_i(str_1, str_2)$ is minimized. As each character is determined in turn, the objective function $\Re$ with respect to the predicate can become negative (or zero).

If strings str_1 and string str_2 have the same values on input x_0, the test generation algorithm does not need to be invoked. We select x_0 and the corresponding input with which the last character of the adjusted variable *adjustr* is increased or decreased by 1 (the remaining input variables are held unchanged) as *ON* and *OFF* test points, respectively, depending on the operator *op* in the string predicate. If str_1 and str_2 are not equal, the corresponding characters of strings str_1 and str_2 are compared from position 0 to $L-1$. That is, an objective function $\Re_i$ is constructed such that $\Re_i = d_i(str_1, str_2) > 0$ for the ith unequal character. Then, an adjustment direction is searched by modifying the ith character of the adjusted variable *adjustr*, denoted by c_i, namely 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 the appropriate direction has been found; otherwise, if there is another input variable, it is selected for adjustment. If all variable

[‖]Nowadays, a general objective function for each predicate is made up of two components—the approach level and the branch distance [17]. The approach level measures how close an input was to executing the target node. The branch distance reflects how close the alternative branch was to being taken when the first component is minimized to zero. In this paper, we aim at the effectiveness of the branch distance component on *ON–OFF* test point generation for string predicate border. Thus, we ignore approach level determination, and manually set the branches before the target predicate to be True or False according to its path condition in our experiments. In further work, the effectiveness of the approach level component will be taken into account.

DOI: 10.1002/stvr

adjustment options have been exhausted unsuccessfully, the *ON–OFF* test point generation has failed for the predicate border. For instance, suppose that only the input variable *instr* is connected with a predicate $strcmp(str_1, str_2) > 0$, and the program implements the function: str_1 ='abc'+*instr*, str_2 = '2334'. In this case, irrespective of the value of the input variable *instr*, str_1 is always greater than str_2. Hence, there is no possible *OFF* test point for the border.

When a good direction is found, the adjusted amount can be boldly increased (e.g. doubled) until either (1) $\Re_i \leq 0$, or (2) $\Re_i$ is not improved, or (3) constraint violation occurs, or (4) c_i' is outside the range 32–126. In the last three cases, we reduce the adjustment amount and the corresponding input is tried again. In the first case of $\Re_i \leq 0$, we obtain two distinct values, referred to as C_{on} and C_{off}, with respect to the adjusted variable *adjustr* on position i, such that C_{on} meets $\Re_i \leq 0$, whereas C_{off} satisfies $\Re_i > 0$. The two values are refined gradually with the help of a temporary variable C_{itemp} whose initial value is C_{off}. Subsequently, we halve the adjusted amount by which C_{itemp} is modified. The code is executed with the new input and $\Re_i$ is evaluated. If $\Re_i$ corresponding to C_{itemp} is negative, then C_{on} takes the value of C_{itemp}; otherwise C_{off} takes the value of C_{itemp}. The process is repeated until the distance $d_i(str_1, str_2)$ has not been diminished. If $d_i(str_1, str_2)$ is equal to 0, the ith character of the adjusted variable *adjustr* has been determined, and the next character, i.e. the $(i+1)$th character, is taken into account. If the adjusted variable *adjustr* ends before position $L-1$, namely *adjustr*[i]='\0' ($i < L-1$), then a space character is added at position i and the next position acts as the string termination, i.e. *adjustr*[i] = ' ' and *adjustr*[$i+1$] = '\0'. The comparison continues until $i = L-1$ or there is no decrease on $d_i(str_1, str_2)$ and $d_i(str_1, str_2) \neq 0$. In the case of $i = L-1$, if $d_{L-1}(str_1, str_2)$ is equal to 0, the current input and the input with the adjusted variable plus or minus 1 on the last character (where other variables are kept unchanged) are selected, respectively, as the *ON* (or *OFF*) test points depending on the operator *op* in the string predicate. As a result, the distance between *ON* and *OFF* test points is only 1, namely the shortest. In the case of $d_i(str_1, str_2) \neq 0$, the $d_i(str_1, str_2)$ is minimized, denoted by $d_{i-\min}$, and we get two refined C_{on} and C_{off} values. We take the C_{on} and C_{off} as the ith character of the adjusted variable *adjustr*, respectively, and keep other characters of the variable *adjustr* as well as other variables unchanged. Thus, we obtain two inputs, selected as the *ON* (or *OFF*) test points, and the distance between them is $d_{i-\min} \times w^{L-i-1}$.

The main algorithm to generate *ON–OFF* test points with respect to a string predicate border associated with a program path is thus described as follows:

For *the* i^{th} *character* c_i *of adjusted input variable*
  Initialize $\Re_i$
  Search an adjustment direction ***Dir***$(\pm)$
If (***Dir*** *is not found*), *exit*
Else *initialize AMOUNT*$\Leftarrow 2$
  Repeat
    $c_i = c_i$ ***Dir*** *AMOUNT*
    Invoking instrumented_program()
    Evaluate $\Re_i$
    If $(\Re_i \leq 0)$
      Obtain two distinct characters C_{on} *and* C_{off} *at position i*
      Repeat *refining the distinct characters*

DOI: 10.1002/stvr

```
                    Until d_i is minimized
                    If (d_i = 0)
                        the ith character is determined
                    Else obtain ON, OFF points, exit
                    Endif
                Else If (ℜ_i decreases)
                        AMOUNT ⇐ AMOUNT ×2   // Enlarge adjusted amount.//
                    Else AMOUNT ⇐ AMOUNT\2   //Lessen adjusted amount.//
                    Endif
                Endif
            Until (AMOUNT ≤ 1)
        Endif
Endfor
```

4.4. An example of ON–OFF test point generation in a string domain

This section gives an example to illustrate the *ON–OFF* test point generation procedure in detail.

Max is a variation of a program in [24]. It prints the lexicographic maximum of its command-line arguments. *Max* has an option: *-ceiling* which provides a ceiling: if the maximum would be larger than this, it is the maximum. If the word after 'max' begins with a '-' and it is not '-ceiling', then an error message will be printed. The author of Reference [24] considers 'repeated option' as an interesting requirement, and corresponding checking is designed into a *for* loop. According to the specification of *Max* program and for convenience, we change the *for* loop into an *if* statement. In addition, some data declaration statements and initialization codes are deleted, e.g. #define BUFSIZE 20, result [BUFSIZE]='\' 0, etc. for simplicity purpose.

As shown in Figure 2, *Max* has two input variables. One is an integer variable *argc*, and the other is a string variable *argv*. Although the program is made up of only a handful of statements, it contains a number of structures, such as numerical predicates, string predicates, compound predicates, ***if–then–else*** statements and ***for*** loop statements. The current values of variables in the predicates are calculated or collected by the program instrumentation technique. An instrumented version of the *Max* program is shown in Figure 3, in which the instrumentation statements are displayed in italics.

Suppose program *Max* is traversed along path {1, 2, 3, 4, 5, 6, 7, 10, 13, 14, 15, 16, 17, 19} on input x: $argc = 4$, argv[1]='$-ceiling$', $argv[2]$='193' and $argv[3]$='$A2$'. The predicate $strcmp(result, ceiling) > 0$) (statement 16) refers to a given string predicate, where *ceiling* is the target string and $argv[3]$ is an adjusted input variable. The instrumented *Max* program is executed, and we gain the current value of the variables *result* and *ceiling* in the predicate on input x, that is, *result* = 'A2', and *ceiling* = '193'.

According to the test generation algorithm, the *ON–OFF* test points for the predicate border are as follows:

ON test point: $argc = 4$, $argv[1]$='$-ceiling$', $argv[2]$='193', $argv[3]$='192'.

OFF test point: $argc = 4$, $argv[1]$='$-ceiling$', $argv[2]$='193', $argv[3]$='193'.

It is clear that the distance between *ON* and *OFF* test point is minimized, which is equal to just 1. The details are as follows.

DOI: 10.1002/stvr

```
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++; /*Skip argument.*/
7                   argc--;       argc--;         }
              else
8               {   fprintf("Illegal option %s.\n",argv[0]);
9                   return(0);    };       }
10    if(argc==0)
11    {         fprintf("At least 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);
 }
```

Figure 2. *Max* Program.

```
record (argc,0,'>',"&&");
record('-',**argv, '=');
if ((argc>0)&&('-'==**argv))
{   record(argv[0],"-ceiling", '!');
    if (!strcmp(argv[0],"-ceiling"))
    …;
  }
record(argc,0,'=');
if(argc==0)
…;
record(argc,0,'>');
for(;argc>0;argc--,argv++)
{   record(argv[0],result, '>', 0);
     if (strcmp(argv[0],result)>0)
        …;
     record(argc,0,'>');
}
record(ceiling,result, '<=', 0);
if (strcmp(ceiling,result)<=0)
          …;
```

Figure 3. Instrumented *Max* program.

DOI: 10.1002/stvr

Table III. (a) Search for distinct characters at position 0 and (b) refine distinct characters at position 0.

AMOUNT	C_0	$C_0^{'}$	$\Re_0$	*Ceiling*[0]	*Dir*
(a)					
1	65	64	15		−
2	64	62	13		
4	62	58	9	49	
8	58	50	1		
16	50	34	<0		
(b)					
AMOUNT	C_{off}	C_{on}	C_{itemp}	$\Re_0$	
16	50	34	$50-8=42$	<0	
8		42	$50-4=46$	<0	
4		46	$50-2=48$	<0	
2		48	$50-1=49$	$=0$	

Table IV. Search for distinct characters at position 1.

AMOUNT	C_1	$C_1^{'}$	$\Re_1$	*Ceiling*[1]	*Dir*
1	50	51	6	57	+
2	51	53	4		
4	53	57	0		

At position 0, *argv*[3][0] = '*A*'. Here, *result*[0] = '*A*', with an ASCII value of 65, and *ceiling*[0] = '1', with an ASCII value of 49. Thus, $\Re_0 = result[0] - ceiling[0] = 16$ and the adjustment direction will be negative. The steps required for the determination of the 0th distinct character are demonstrated in Table III(a).

When $\Re_0 < 0$, we get two distinct characters whose ASCII values are 50 and 34, respectively. The two distinct characters are refined gradually until $\Re_0 = 0$ (shown in Table III(b)). Here, C_{itemp}=49. Thus ,*argv*[3][0] is set to 49, and thus the 0th character of input variable *argv*[3] has been successfully determined; at this point, *argv*[3] = '12'.

Then, we compare the 1st character of variables *result* and *ceiling*. At position 1, *argv*[3][1] = '2'. Here, *result*[1] = '2', whose ASCII value is 50, and *ceiling*[1] = '9', whose ASCII value is 57. Thus, $\Re_1 = ceiling[1] - result[1] = 7$, and the adjustment direction will be positive. The steps required for the determination of the 1st distinct character are shown in Table IV.

When $C_1^{'} = 57$, we derive $\Re_1 = 0$. Thus, it is of no value for the process to refine two distinct characters. We thus set *argv*[3][1] = 57, and hence the 1st character of input variable *argv*[3] has been obtained. At this point, *argv*[3] = '19'. We now continue to compare the next character.

At position 2, *argv*[3][2] = '\0', but *ceiling*[2] $\neq$ '\0'. Let *argv*[3][2] = ' ', and *argv*[3][3] = '\0'. Here *result*[2] = ' ', whose ASCII value is 32, and *ceiling*[2] = '3', whose ASCII value is 51. Thus $\Re_2 = ceiling[2] - result1[2] = 19$, and the adjustment direction will be positive. The steps required for the determination of the 2nd distinct character are shown in Table V.

DOI: 10.1002/stvr

Table V. (a) Search for distinct characters at position 2 and (b) refine distinct characters at position 2.

AMOUNT	C_2	$C_2^{'}$	$\Re_2$	*Ceiling*[2]	*Dir*
(a)					
1	32	33	18		
2	33	35	16		
4	35	39	12	51	+
8	39	47	4		
16	47	63	<0		
(b)					
AMOUNT	C_{off}	C_{on}	C_{itemp}	$\Re_2$	
16	47	63	$47+8=55$	<0	
8		55	$47+4=51$	$=0$	

When $C_{itemp}=51$, $\Re_2=0$. Let *argv*[3][2]=51; then *argv*[3]='193'. The algorithm terminates since *ceiling*[3]='\0'. According to the operator of the predicate, the current input is selected as the *OFF* test point, i.e. the *OFF* test point is *argc*=4, *argv*[1]= '*-ceiling*', *argv*[2]='193', *argv*[3]='193'. The *ON* test point is the same input as the last character of the adjusted variable $argv[3]$ decremented by 1, i.e. *argv*[3][2]=*argv*[3][2]−1, while the other input variables remain unchanged. Therefore, the *ON* test point is *argc*=4, *argv*[1]='*-ceiling*', *argv*[2]='193' and *argv*[3]='192'. It can be seen that the distance between the *ON* and *OFF* test points obtained in this way is minimized.

4.5. Some explanations for the string test generation algorithm

As mentioned above, $\varphi(str)$ may be very large for long strings. However, integer overflow can be avoided because our implementation of the algorithm compares and searches for one character at a time using a single character distance function.

Although generally speaking, greedy heuristic algorithms can fail if a local minimum is encountered, this cannot happen in our case. The statement for this is as follows. We have analysed 163 library functions in C Language [25] and found 462 simple predicates, of the form $str\_cmp(str_1, str_2)$ *op* 0 or E_1 *op* E_2, where str_cmp is a string comparison function, E_1 and E_2 are arithmetic expressions and *op* is one of $\{<, \leq, =, \neq, >, \geq\}$. In 388 of these predicates (84% of the total), one of E_1 and E_2 (or string str_1 and str_2) is constant (or string literal). In the remaining 74, there are 47 predicates (10%) for which each variable is related to a different input variable. The two variables in the other 27 predicates (only 6% of the total) are involved with one input variable. Consequently, we can draw the conclusion that for simple string predicates, in most cases, one of string str_1 and string str_2 is irrelevant to the adjusted variable. Let c_i represent the ith character of the adjusted variable and $L=\max(L_1, L_2)=L_2$. Thus, at position i, we have $\Re_i=|str_1[i]-str_2[i]|$, $(0\leq i\leq L_1-1)$ or $\Re_i=str_2[i]$, $(L_1\leq i\leq L-1)$. If str_1 is independent of the adjusted variable, then $str_1[i]$ is not connected with c_i and can be thought of as a constant, represented by M, whereas $str_2[i]$ is a function of c_i, denoted as $\vartheta(c_i)$. Accordingly, the objective function $\Re_i$ can be expressed as $\Re_i=|M-\vartheta(c_i)|$ or $\Re_i=\vartheta(c_i)$. If str_2 is independent of the adjusted variable, then

DOI: 10.1002/stvr

$\Re_i = |\vartheta(c_i) - M|$, $(0 \le i \le L_1 - 1)$ or $\Re_i = M(L_1 \le i \le L - 1)$. If $\Re_i$ is a constant, no matter whether $c_i' = c_i + 1$ or $c_i' = c_i - 1$, c_i' does not result in a better $\Re_i$ value than c_i. In this case, our algorithm selects another input variable for adjustment.

Hence, the objective function $\Re_i$ is a monotonically increasing or decreasing function, i.e. the effect of adjusting each character is not restricted to a localized region of $\Re_i$. The objective function $\Re_i$ can reach negative values or zero so that each character of the adjusted variable is determined in turn. As a result, the function $\Re$ does not suffer from the local minimum problem.

5. EMPIRICAL ASSESSMENT OF THE STRING ON–OFF TEST GENERATION

In order to investigate and illustrate the effectiveness of the string *ON–OFF* test generation, we have conducted a substantial number of experiments for ten C programs of unit and module-level containing string predicates, obtained from books or Web source [24,26], and an instrumented version of each of them is constructed to compute the current values of the variables in the predicates. The string predicates can be simple or compound. A compound predicate can be decomposed easily into two or more simple predicates. In this case, we separately generate *ON–OFF* test points for each simple predicate border, and add a new constraint arising from the remaining part of the compound predicate. The constraint requires that the Boolean result of the remaining part should have no influence on the result of the compound predicate; i.e. if the result of the simple predicate P_{simp} is true, then the result of the compound predicate P_{comp} is also true. The decomposition is illustrated by the following example.

Example. Assume that a compound predicate P_{comp} contains five simple predicates *P1*, *P2*, *P3*, *P4* and $P5$ as follows: $P_{comp} = (P1\ or\ P2)\ and\ (P3\ or\ P4)\ and\ P5$. All five simple predicates should be separately tested. Now consider $P1$, then the remaining part of P_{comp} should take $P2 = \mathrm{F}$(Flase), $(P3\ or\ P4) = \mathrm{T}$(True) and $P5 = \mathrm{T}$. Thus, the result of $P2$, $P3$, $P4$ and $P5$ can be:

$P2$	$P3$	$P4$	$P5$
F	T	T	T
F	F	T	T
F	T	F	T

As described in Table VI, the total counts of LOC and the number of string predicates (NOSP) are listed for each program. The *Max* program has four string predicates, the *Booksystem* program 4 and the *Findstring* program 3, denoted by the corresponding program name following '*-pre*' and the serial number of the predicates, e.g. the fourth predicate of the *Max* program is signified by *Max-pre4*. Each of the other programs has a string predicate, denoted by the corresponding program name, e.g. predicate *Student* corresponds to program *Student*. A target string in the predicate under the test is the string with which input will match. In the predicates *Max-pre1*, *Max-pre2*, *Password*, *Random*, *Book-pre1*, *Book-pre2*, *Book-pre3*, *Book-pre4*, *Find-pre1*, *Find-pre2* and *Find-pre3* the target strings are literals containing 1, 8, 5, 4, 4, 6, 6, 4, 5, 5 and 3 characters, respectively. For example, the target string in predicate *Max-pre2* is equal to literal '*-ceiling*'. The target strings in the remaining predicates are string variables, such as *result* in predicate *Max-pre3*. If the target string is not a literal, in order to simplify the task, we assume that it contains 5 characters, or that its length is not larger than 10.

DOI: 10.1002/stvr

Table VI. Experimental programs description.

Name	Program description	LOC	NOSP
Max	Search for the max string from a set of strings supplied by the user	53	4
Password	Check if a password is right or wrong	15	1
Random	A little game with random information	31	1
Concate	Concatenate and compare two strings entered by the user	63	1
Student	Creates a student list. User can add, display, delete names, and apply bubble sort and binary search	122	1
Booksystem	Record whether books are borrowed or returned.	56	4
Bubble	A program to sort strings	53	1
Findstring	A program to find a string among three strings	26	3
Notebook	A notebook management program	21	1
Inventory	A inventory management program	43	1

DOI: 10.1002/stvr

In our experiments, 24 initial input sets are designed for each string predicate, each consisting of 20 inputs that vary in length from 1 to 20 characters, and all character values are limited in the range between 32 and 126. In total, $24 \times 20 = 480$ trials are conducted for each string predicate. The maximum string length of 20 characters is used to create a large search space. The search is conducted to generate *ON–OFF* test points for one string predicate border at a time.

Dynamic test data generation is a heuristic process. When a new input is created, the instrumented program has to be executed again in order to evaluate its objective function. The cost of the *ON–OFF* test generation algorithm depends mainly on the number of times that the objective function must be evaluated, i.e. the number of times of the instrumented program is executed. Thus, in this paper, the performance of the *ON–OFF* test point generator is defined as the number of evaluations of the objective function. For each string predicate in the programs listed in Table VI, the average numbers of evaluations in the 480 trials are shown in Figure 4. Here, *String literal* implies that the target string in the predicate under test is a literal. *String of fixed length* indicates that the target string is a variable composed of 5 characters, and *String of varied length* points out that the length of the target string is a random integer between 1 and 10, but each character is chosen at random. These results show that the number of evaluations of the objective function corresponding to the predicates *Max-pre3*, *Max-pre4*, *password*, *Concate*, *Student*, *Bubble*, *Find-pre1*, *Find-pre2*, *Notebook* and *Inventory* is approximate, even though some of the target strings are literals with 5 characters, and the others are treated as variables whose lengths range from 1 to 10 or are fixed at 5. For the target string in predicate *Student*, we count the average number of the evaluations in two cases, namely *fixed length* and *varied length*.

To obtain a more detailed insight, we analyse the experimental results using the statistical analysis tool SPSS. The statistical descriptions are shown in Tables VII(a) and (b). In Table VII(a), corresponding to the greyish bar for *Password, Find-pre1* and *Find-pre2* and the black bars for *Concate Student, Bubble* and *Notebook* in Figure 4, there are only slight differences in their mean numbers of evaluation of the objective function as well as their standard deviations, and their standard deviations are small for only about 6.67. In Table VII(b), with reference to the white bars for *Max-pre3*, *Max-pre4*, *Student* and *Inventory* in Figure 4, there are only trivial differences in their mean numbers of the evaluation as well as standard deviations, but their standard deviations are larger for about 22.18. In addition, the number of evaluations ranges between 1 and 102, whereas the number varies only between 16 and 58 for the experiments in Table VII(a). This is easy to explain since the target strings in the predicates in Table VII(b) differ between 1 and 10 in length, whereas

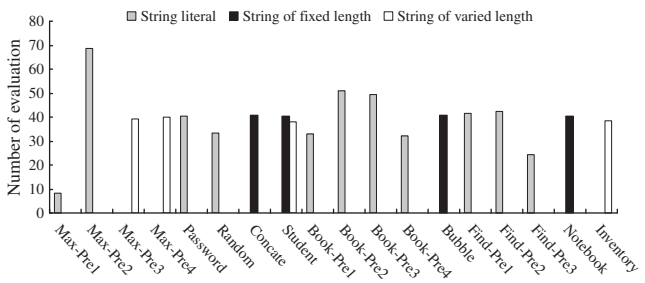

Figure 4. Predicates under test.

DOI: 10.1002/stvr

Table VII. (a) Stastistical results of target string with 5 characters and (b) statistical result of target string with varied character number.

Predicate under test	*N*	Minimum	Maximum	Mean	Std. deviation
(a)					
Password	480	17	56	41.2938	7.23590
Concate	480	24	58	40.7979	6.39405
Student	480	17	57	40.5750	6.46030
Bubble	480	23	56	40.5896	6.42024
Find-pre1	480	16	54	41.6271	6.82655
Find-pre2	480	21	56	42.4500	7.45548
Notebook	480	19	55	40.5771	6.67234
(b)					
Max_pre 3	480	1	94	39.3604	22.18214
Max_pre 4	480	1	94	39.8438	21.89076
Student	480	1	102	38.1042	22.23404

the lengths of the target string in the predicates in Table VII(a) are a constant 5. However, the mean numbers of evaluations in Table VII(b), about 39.36, are close to the means in Table VII(a), about 40.79. Therefore, a conclusion can be drawn from the result that there is little connection between the number of evaluations and the value of the target string as well as the program under test.

It is further observed that the number of the evaluation has a close correlation with the length of target string. However, what is this relationship? Is there a constant upward or downward trend that follows a straight-line pattern or a curved pattern? To address these questions, we apply a regression analysis to the data of the *Concate* predicate. The plot of the regression line and the actual data points is illustrated in Figure 5. The scatterplot exhibits a very strong linear relationship between the number of the evaluation and the length of the target string. The formula for the linear regression is

$$Number\_of\_Evaluation = -2.14 + 8.45 * Length\_of\_String$$

The coefficient of determination *R-Square* is 0.96 (96%), which is substantial. That is to say, the regression model works very well. For example, if the target string includes 5 characters, from the regression formula, the number of the evaluation would be $-2.14+8.45*5=40.11$, whereas the actual mean, from Table VII(a), is 40.80 with respect to predicate *Concate*. As another example, for the target string literals in the predicates *Max-pre1, Random, Password* and *Max-pre2*, comprising 1, 4, 5 and 8 characters, respectively, the means of 480 trials on each string predicate are compared with the values from the regression model. The result is displayed in Figure 6. It is clear that the actual means of the number of the evaluation are very close to the values from the regression model. Therefore, the regression model can be used to estimate or predict the number of the evaluation when the length of the target string in the predicate under test is known.

The function of test point generation is to find an input to match the target string in the predicate under test. It is evident that the initial input affects the efficiency of test data generation. This brings us to the question posed earlier: what relationship exists between the number of the evaluation and the length of initial input string? Furthermore, how should the characters in the initial input be selected to enhance the performance of the test generation algorithm? In order to answer the

DOI: 10.1002/stvr

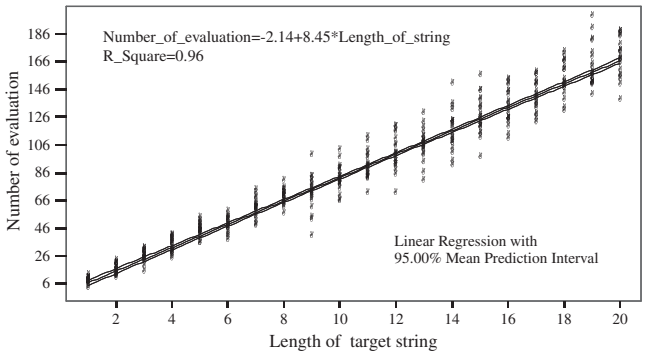

Figure 5. Correlation of number of evaluation versus length of target string for *Concate* predicate.

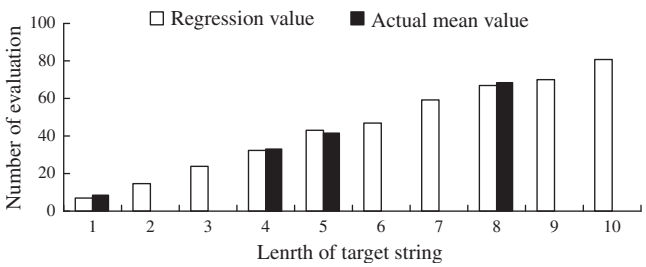

Figure 6. Compare regression and actual mean values.

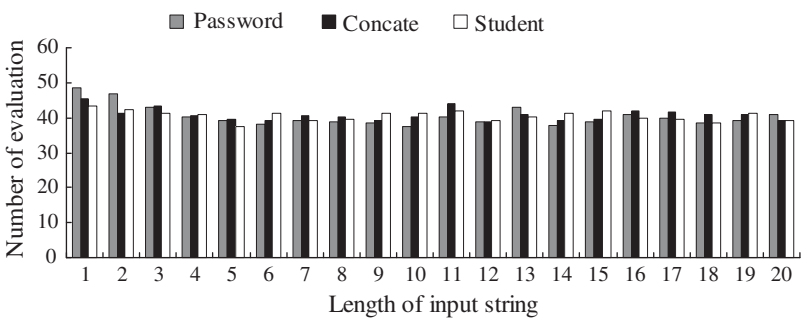

Figure 7. Number of evaluation versus length of input string.

first problem, we analyse the number of the evaluation versus different lengths of initial input string, for predicate *Password*, *Concate* and *Student*, where the target strings include exactly 5 characters. The result is shown in Figure 7. It can be observed that there are no distinct differences in the number of the evaluation with respect to the length of initial inputs ranging from 1 to 20. A more detailed statistical description can be found in Table VIII corresponding to predicate

DOI: 10.1002/stvr

Table VIII. Statistical description of Concate predicate.

Length of string	*N*	Minimum	Maximum	Mean	Std. deviation
1	24	34	55	45.2917	5.3445
2	24	26	51	41.2500	6.9673
3	24	31	56	43.2500	6.0163
4	24	32	49	40.5417	4.8810
5	24	24	52	39.5417	7.2049
6	24	27	54	39.1667	7.0444
7	24	27	53	40.5417	7.7234
8	24	26	54	40.0833	7.5695
9	24	29	53	39.1667	5.9466
10	24	27	54	40.3750	6.8829
11	24	35	58	43.9583	5.8717
12	24	28	52	38.9167	6.2618
13	24	30	50	40.9167	4.6054
14	24	27	53	39.1250	6.5628
15	24	28	51	39.3750	6.3439
16	24	31	51	41.9583	6.1040
17	24	32	51	41.5417	6.0215
18	24	31	51	40.8750	5.9659
19	24	28	51	40.7917	5.5949
20	24	24	48	39.2917	5.9453

Concate. The minimum and maximum numbers of the evaluations vary in the ranges 24–34 and 48–58, respectively, whereas the mean varies only between 38.92 and 45.29 when the length of the initial input is varied from 1 to 20. For predicate *Password*, whose target string is a literal with 5 characters, there is an obvious increase in the number of the evaluation when initial input has only a few characters, especially only 1 or 2 characters. The trend is clearer in Figure 8, where the heaviest dotted line indicates the mean number of the evaluation in predicate *Password*. This can be explained by noting that excess characters in the initial input are thrown away, whereas if the input is too short, additional space characters need to be added. A similar trend is found in predicates *Max-pre2* and *Random*, *Book-pre1*, *Book-pre2*, *Book-pre3*, *Book-pre4*, *Find-pre1*, *Find-pre2* and *Find-pre3*, shown in Figure 9, where the target strings in these predicates are literals. Therefore, we can derive the proposition that the initial input should not be shorter than the target string in the predicate under test.

To address the second problem, i.e. how to select the characters in initial input, 480 initial inputs are run on each string predicate, consisting of 24 initial input sets, each comprising 20 input strings ranging in length from 1 to 20. The first input set employs only the space character (the lowest printable ASCII value); i.e. the 20 inputs contain 1–20 space characters, respectively. Similarly, we create the second, third and fourth input sets composed of only, respectively, '*', '<' and '∼' characters (the last being the highest printable ASCII value), which are referred to as *fixed characters* in the following section. The remaining 20 input sets are made up of random characters limited in range from space to the '∼' character. The effect of different characters on the number of the evaluation with respect to predicate *Password* is demonstrated in Figure 8. In this figure, the two fine dotted lines indicate the number of the evaluation when initial inputs comprise,

DOI: 10.1002/stvr

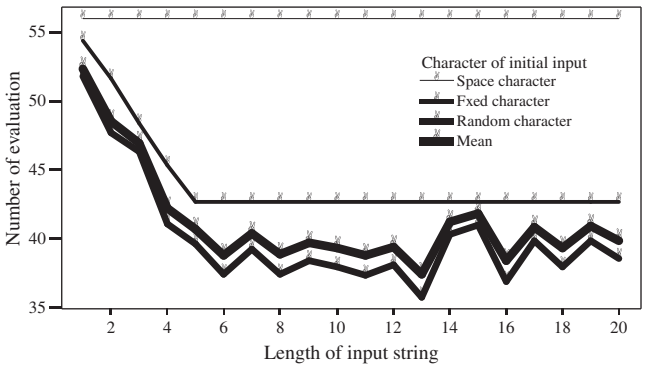

Figure 8. Compare different initial character *Password* predicate.

respectively, space and *fixed* characters, the heavier dotted line shows the mean when initial inputs consist of random characters, and the heaviest dotted line shows the mean of each group of 24 inputs corresponding to a certain length of input string. It is found that the number of the evaluation is the largest when all initial inputs are composed of space characters. The least number of the evaluation results from initial inputs in which all characters are generated randomly. In addition, the initial inputs containing only space characters have the same number of the evaluation, even if their length changes from 1 to 20. For the initial inputs with *fixed characters*, the number of the evaluation decreases in tandem with the increase in the length of the input string until the length is equal to that of the target string, here 5 for the predicate *Password*, and then the number of the evaluation remains constant. For the initial inputs with random characters, the number of the evaluation also diminishes as the length of the input string increases. However, when the length of inputs is larger than that of the target string, the number of the evaluation fluctuates, but no more so than with the *fixed character* input. The same behaviour is seen for the predicates with string literal in Figure 9. As a result, we hypothesize that using characters generated randomly in initial inputs produces more effective string test data than *fixed* and space characters, when target strings in the predicate under test are literals.

For the predicates *Concate*, *Student*, *Bubble* and *Notebook* for which the target strings are 5 characters in length, but not literal, the effect of different initial characters on the number of the evaluation is shown in Figure 10. Evidently, initial inputs comprised of random characters result in a lower number of the evaluation on average, and the mean number of the evaluation fluctuates less than the number for inputs consisted of space or *fixed* characters. Figure 11 shows a similar result for the predicate *Max-pre3*, *Max-pre4*, *Student* and *Inventory*, whose target strings range from 1 to 10 characters. Also, the variations in the mean number of the evaluation with respect to the target strings including 5 characters are compared with those relative to the target strings ranging between 1 and 10 characters. The result is displayed in Figure 12. The means in the two cases are almost the same, as shown in Figure 4, Table VII(a) and (b), although the fluctuations in the mean are larger when the length of the target strings varies. Thus, we can draw the conclusion that the characters in the initial input string should be generated randomly in order to enhance the performance of string test data generation, no matter whether the target strings involved in the predicate under test

DOI: 10.1002/stvr

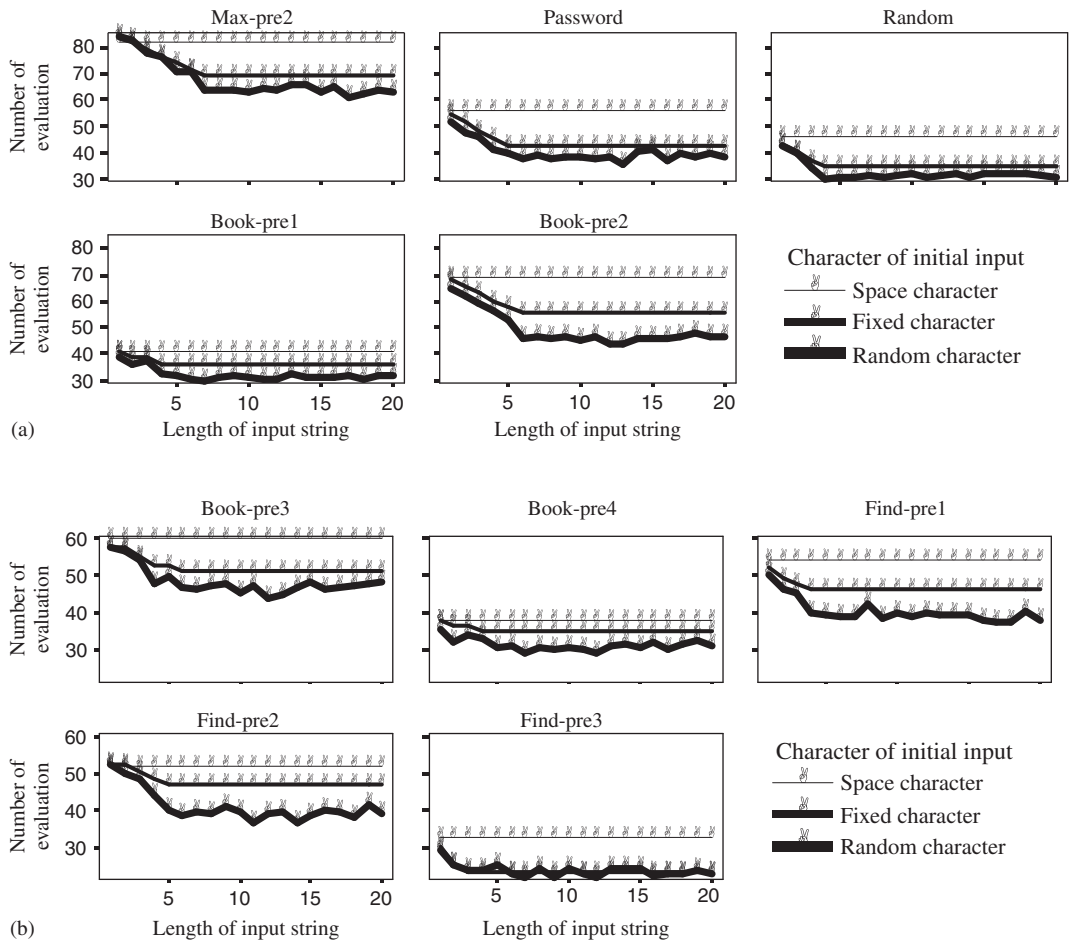

Figure 9. (a) Compare different initial characters for predicates with literal and (b) compare different initial characters for predicates with literal.

are literal, fixed length or varied length. Indeed, it is not advisable that the initial input string be comprised of any single character, especially the characters with minimum and maximum printable ASCII values.

However, if the target string in the predicate under test is a literal that contains only one character, such as for predicate *Max-pre1*, the above conclusions are invalid. Figure 13 compares the effect of different initial characters on the number of the evaluation for predicate *Max-pre1*. Four hundred and eighty trials show that the initial character generated randomly yields a larger number of the evaluation than fixed and space characters. This indicates that the space character may be a good choice for initial input relative to a target string containing only a single specified character.

DOI: 10.1002/stvr

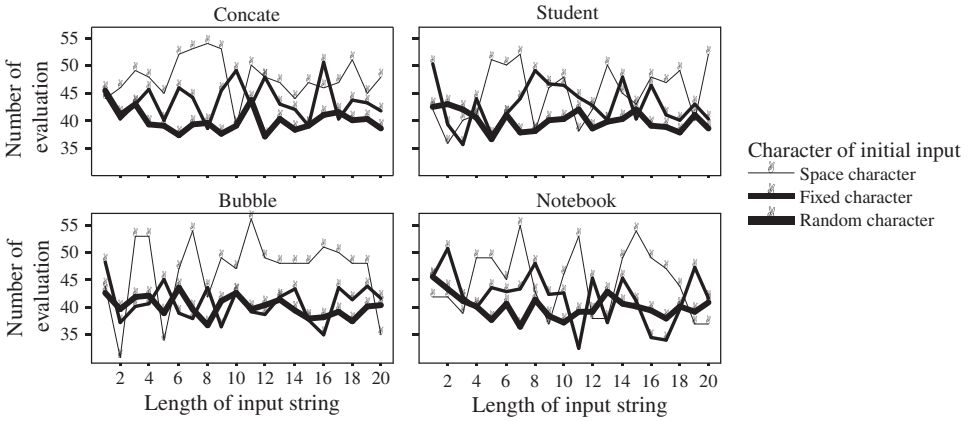

Figure 10. Compare different initial characters for predicates with fixed length.

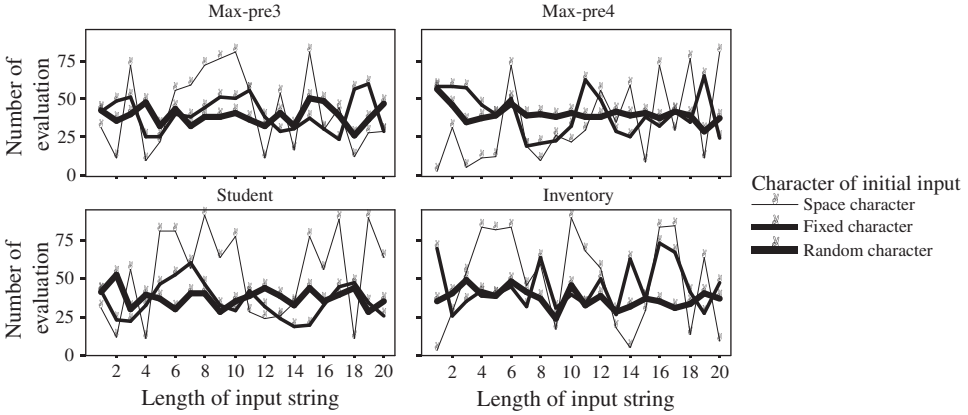

Figure 11. Compare different initial characters for predicates with varied length.

6. CONCLUSION

The objective of domain testing is to detect domain errors in programs. However, current work in domain testing strategies has been limited largely to programs in which string predicates are not taken into consideration. The same weakness is found in many currently available test data generation systems. In this paper, we have presented a novel approach for the automatic generation of *ON–OFF* test points for string predicate borders associated with program paths, and have developed a corresponding test data generator.

In order to investigate and illustrate the effectiveness of our string *ON–OFF* test generation approach, a number of experiments have been conducted on a set of programs containing string predicates. Four hundred and eighty trials have been conducted for each string predicate under

DOI: 10.1002/stvr

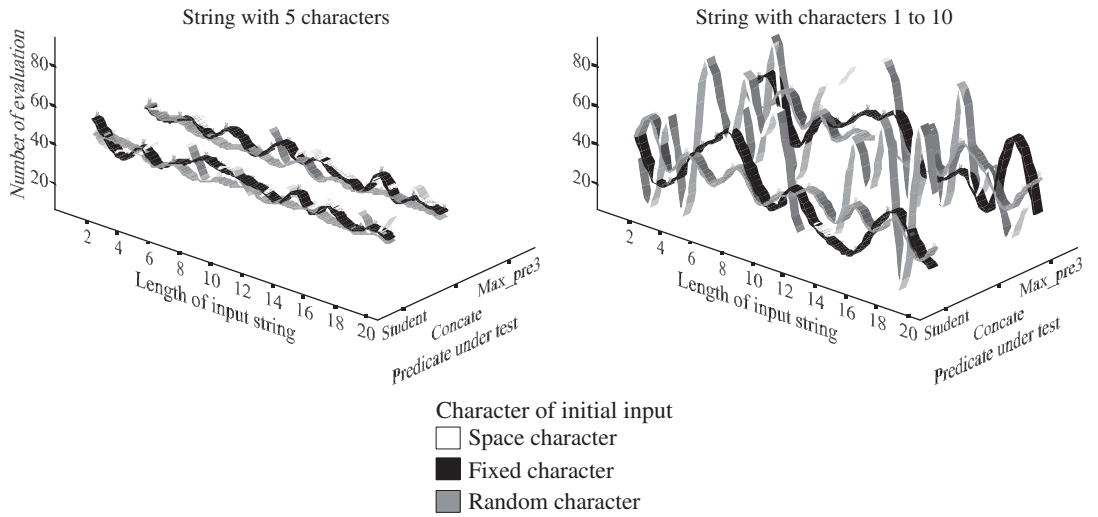

Figure 12. Compare predicates with fixed and varied length.

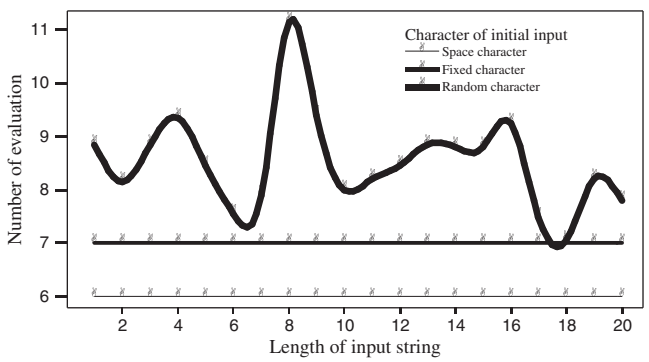

Figure 13. Compare different initial characters for predicates *Max-pre1*.

test, and the experimental results were analysed in detail using statistical analysis tool SPSS. We conclude that there is a strong linear relationship between the performance of the test generator and the length of the target string in the predicate under test, and this relationship can be used to estimate or predict the number of the evaluation of the objective function in order to match the target string when the length of the target string is known. Moreover, with initial inputs not shorter than the target string and with characters generated randomly, the performance of our string test data generation algorithm can be enhanced. In summary, the results show that the methodology is promising and effective in string test data generation for detecting domain errors.

DOI: 10.1002/stvr

ACKNOWLEDGEMENTS

Shanshan Lv conducted extensive experiments and analysis for this paper. Here we express our heartfelt appreciation. The work described in this paper was supported by the National Natural Science Foundation of China under Grant No.60473032; Beijing Natural Science Foundation under Grant No.4072021; and the Hong Kong Research Grants Council, under Grant No.CUHK4150/07E.

REFERENCES

1. Lyu MR, Rangarajan S, Moorsel AP. Optimal allocation of test resources for software reliability growth modeling in software development. *IEEE Transactions on Reliability* 2002; **51**(2):183–192.
2. Offutt AJ, Jin ZY, Pan J. The dynamic domain reduction approach to test data generation. *Software Practice and Experience* 1999; **29**(2):167–193.
3. Jeng B, Forgacs I. An automatic approach of domain test data generation. *The Journal of Systems and Software* 1999; **49**:97–112.
4. White LJ, Cohen EI. Domain strategy for computer program testing. *IEEE Transactions on Software Engineering* 1980; **6**(3):247–257.
5. Clarke LA, Hassell J, Richardson DJ. A close look at domain testing. *IEEE Transactions on Software Engineering* 1982; **8**(4):380–390.
6. Zeil SJ, Afifi FH, White LJ. Detection of linear errors via domain testing. *ACM Transactions on Software Engineering and Methodology* 1992; **1**(4):422–451.
7. Jeng B, Weyuker EJ. A simplified domain-testing strategy. *ACM Transactions on Software Engineering and Methodology* 1994; **3**(3):254–270.
8. Hajnal A, Forgacs I. An applicable test data generation algorithm for domain errors. *ISSTA'98*, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, FL, U.S.A., March 1998; 63–72.
9. Jeng B. Toward an integration of data flow and domain testing. *The Journal of Systems and Software* 1999; **45**:19–30.
10. Alshraideh M, Bottaci L. Search-based software test data generation for string data using program-specific search operators. *Software Testing Verification and Reliability* 2006; **16**(3):175–203.
11. Howden WE. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering* 1976; **2**(3):208–215.
12. Michael CC, McGraw G, Schatz MA. Generating software test data by evolution. *IEEE Transactions on Software Engineering* 2001; **27**(12):1085–1110.
13. Gotlieb A, Petit M. Path-oriented random testing. *Proceedings of the First International Workshop on Random Testing* (*RT'06*), Portland, U.S.A., July 2006; 28–35.
14. Gareth L, Morris J, Parker K, Gary A. Using symbolic execution to guide test generation. *Software Testing*, *Verification and Reliability* 2004; **15**(1):41–61.
15. Coward PD. Symbolic execution and testing. *Information and Software Technique* 1991; **33**(1):229–239.
16. Korel B. Automated software test data generation. *IEEE Transactions on Software Engineering* 1990; **16**(8):870–879.
17. McMinn P. Search-based software test data generation: a survey. *Software Testing*, *Verification and Reliability* 2004; **14**(2):105–156.
18. Harman M, Hu L, Hierons RM, Wegener J, Sthamer H, Baresel A, Roper M. Testability transformation. *IEEE Transactions on Software Engineering* 2004; **30**(1):3–16.
19. Mansour N, Salame M. Data generation for path testing. *Software Quality Journal* 2004; **12**:121–136.
20. Forgács I, Bertolino A. Feasible test path selection by principal slicing. *Proceedings of the 6th European Conference on Foundations of Software Engineering*, Zurich, Switzerland, 1997; 378–394.
21. Peres LM, Vergilio SR, Jino M, Maldonado JC. Path selection in the structural testing: Proposition, implementation and application of strategies. *Proceedings of the International Conference of the Chilean Computer Science Society* (*SCCC '01*), Chile, 2001; 240–246.
22. Available at: http://www.nist.gov/dads/HTML/HammingDistance.html [6 August 2008].
23. Zhao R, Lyu MR. Character string predicate based automatic software test data generation. *Proceedings of the 3rd International Conference on Quality Software* (*QSIC 2003*), Dallas, TX, November 2003. IEEE Computer Society Press: LosAlamitos, CA, 2003; C255–C263.
24. Marick B. *The Craft of Software Testing*. PTR Prentice-Hall: New Jersey, 1995; 260.
25. Available at: http://ftp.gnu.org/gnu/glibc [6 August 2008].
26. Planet Source Code. Available at: http://www.planet-source-code.com/ [6 March 2007].

DOI: 10.1002/stvr