Test Data Generation for C Programs with String-handling Functions*

Hui Ruan[1,2], Jian Zhang[1] and Jun Yan[1]
[1]State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
[2]Graduate University, Chinese Academy of Sciences
{ruanh, zj, yanjun}@ios.ac.cn

Abstract

There are many test generation methods, but few of them consider the character strings. This paper proposes a method to generate test data for C programs with character strings and character string function calls, which is based on path oriented testing. Each character variable is viewed as an integer variable with the restriction that the value should be between 0 and 255. A character string is viewed as an array of characters with a predefined fixed length. Many commonly used character library functions are modeled by formulae in predicate logic with assignment statements. The model is then used to replace the function call in the program path, which will be solved by a path analysis tool to generate the test data. A prototype tool called StrGen is developed to illustrate the feasibility of this method. The results of some examples also show that this method is feasible and very efficient.

1. Introduction

There are many test generation methods, some of them use the static method [15, 16], while others are based on dynamic method [6, 7, 17]. Recently, there are also some researchers who combine both of these two methods [11, 12]. But few of them consider the character string variable or character string[1] function call. In our previous work, a method and an automatic tool to analyze the feasibility of program path are described in [15]. But it can only process variables of certain data types such as integer, real and array of them with fixed length. In [14], the pointer variable is also handled. And a tool called SimC is presented in [13], which can generate the test cases for programs written in a subset of ANSI C. But none of them supports the string variable or string function call.

In this paper, we extend our early work so that the string variable and string function call can both be handled. Different from the method used in [17, 18], which is based on function minimization and needs to execute the tested program, our method is based on static analysis as well as symbolic execution and constraint solving techniques. We model the string function by logic formulae with assignment statements, and expand the model as a set of sub-constraints, then use the sub-constraints to replace the function call in the program path. We also implemented a prototype tool called StrGen to show the feasibility and efficiency of our method.

The remainder of this paper is organized as follows. In the next section, we briefly review some related work. Section 3 describes the main techniques of how to generate test data for character string with string-handling function call in detail. The results of some examples are presented in section 4. Then section 5 gives some discussion. Finally, we conclude the paper in section 6.

2. An overview of related work

In this section, we review some related work, including different methods to generate the test data, symbolic execution and constraint solving techniques, and some work done on test data generation for string-handling programs.

2.1. Test data generation

By test data generation, we mean, given a program, to generate some input for all the variables, such that the test suite will satisfy some criteria. There are many test data generation methods, and a good survey can be found in [3, 8, 10]. Among these methods, the path-oriented test generation is the most commonly used one, which can be divided into two approaches: *static method* and *dynamic method*. The difference between these two approaches lies

*This work is partially supported by the National Natural Science Foundation of China (NSFC) under grant number 60633010 and the National High-tech R&D (863) Program under grant number 2006AA01Z402.

[1]In this paper, we use *string* to represent *character string* for simplicity.

978-0-7695-3249-3/08 $25.00 © 2008 IEEE
DOI 10.1109/TASE.2008.25

in how the program is executed. The former one uses *symbolic execution* [1, 2, 5], while the latter one uses *actual execution* [3] or so-called *concrete execution* [11]. Some researchers [11, 12] also combine the *static method* and the *dynamic method* together.

In the dynamic method [6, 7, 17], one first executes the program according to some initial input, and then uses some heuristic techniques to adjust the input until the program will execute along the given path. Many different heuristic techniques can be used here, such as function minimization [6, 17], genetic algorithm [7], simulated annealing algorithm [7], and some other stochastic search algorithms in artificial intelligence. Although it is easy to process complicated programs, there are some other problems with this method. As the author of [6] said,

> *"One problem is its very limited ability to detect path infeasibility. If the selected path is infeasible and can not be detected, a large number of attempts can be performed before the search procedure terminates and a lot of effort can be wasted."*

What's more, they may fail to find a test case even when one exists [12].

In the static method [15, 16], one uses *symbolic execution* to "execute" the program path and collect the so-called *path condition*, which is composed of arithmetical constraints and logical expressions. Then, *constraint solving* technique is used to solve the path condition. Finally, one can get the input value for the variables when the path is feasible, or conclude that the path is infeasible if no input value can be found to satisfy the path condition. The infeasible path can be detected in the early stage while using this method, and one needn't actually execute the program .

Some researchers [11, 12] combine the above two approaches, and call the new method *concolic testing*. Each time after they select the test data, they get a path predicate by executing the instrumented code, and use constraint solving techniques to find the next test data. The initial test data is selected randomly. The authors of [4] also use this *concolic* method to generate test input for Database-Driven Application. They generate constraints on strings according to the SQL queries, and then solve the constraints. But, they also show that the satisfiability problem for string constraints is PSPACE [4].

In this paper, we use the static method. That is, we first analyze the input program, translate the source code into some kind of intermediate representation, and get some program paths according to some *coverage criteria* [9, 20]. Then we use symbolic execution and constraint solving techniques to collect and solve the path condition and then generate the test data for each of the selected paths. What we focus on in this paper is the test data generation for the string variables in the presence of string library function calls, which was not considered in our previous works [13, 14, 15] and many other works on test data generation.

2.2. Symbolic execution and constraint solving

Symbolic execution [1, 2, 5, 11] is a useful technique for verification and testing, and has been studied by many researchers during the past 30 years [16]. Different from actual execution, in which one runs a program with concrete values, in symbolic execution, we "run" the program with symbolic values, that is, symbols with some unknown values. Since there may be branch statements in the program, the result of symbolic execution is not a single path, but an *execution tree* [5]. What's more, the existence of some iteration statements will make the execution tree infinite. This brings on much more difficulties for us to symbolically execute the whole program. Instead, we can use symbolic execution on one program path. Such a path has only assignment statements and logical expressions but no branch or iteration statements. After we symbolically execute a program path, we can collect a set of arithmetical equations and logical expressions, called the *path condition*. After solving the path condition, we'll get the concrete values for the input variables, and these values can be used as test data.

In order to solve the path condition, one may use the *constraint solving* techniques, which have been studied by many researchers in both artificial intelligence and operations research communities [16]. The path condition obtained from symbolic execution can be a set of complicated constraints, each of which has both arithmetic and Boolean operators.

In our previous work [15], a tool called PAT can be used to solve these constraints, if only the arithmetic constraints are linear, that is, the arithmetic constraints are linear equations or linear inequations. Given a program path which has only assignment statements and conditional expressions, PAT can symbolically execute the path, collect and solve the path condition. PAT can decide whether that condition is satisfiable. It can also generate the test data when the path condition is satisfiable.

2.3. Character strings

There are many test data generation methods, but few of them consider character strings or string function calls. To the best of our knowledge, only some work done by Zhao et al. in [17, 18, 19] are related to strings. But they used the dynamic methods. Function minimization is also used to generate the test data to satisfy the character string predicate. As we said before, this method needs to execute the tested program and can not detect infeasible paths efficiently. What's more, one needs to map a

character string to an integer value. The method used in [17], is to give a weight for each element of the string. For example, a character string "aa", which has only two elements, is transformed into a number which has the value $97 \times 128 + 97 = 12513$, where 97 is the ASCII code of the character 'a'. When the string is very long, the integer mapped from that string is very big. And these big numbers are very difficult to process.

3. Test generation for character string

Different from the work in [17, 18, 19], we use static method. We extend our previous work [13, 15, 16], in which the character string variables were not supported. A prototype tool called StrGen is implemented based on symbolic execution and constraint solving techniques. It can generate test data for programs including string variables and string function calls. Different from our previous work, we focus on the character strings and string-handling function calls in this paper.

In this section, we first introduce our test generation framework using static analysis and constraint solving techniques. Then, we will focus on the test data generation for character string with the string-handling function.

3.1. Test generation framework

In order to generate the test data for the input programs, we use the following framework:

1. Analyzing the program code and creating abstract syntax tree (AST), translating the AST into an intermediate representation (IR) and building up the control flow graph (CFG).

2. Generating a set of program paths according to some criteria such as path coverage [20], using the traditional graph traversal methods, such as depth-first search (DFS) or breadth-first search (BFS).

3. Using a path feasibility analysis tool to solve the path condition and generate the test data.

In the first step, we parse the input program and build up the CFG. The IR we used here is like the one we used in our previous work [13]. Different from our previous work, the source program we accepted can be written in GCC extended C language, while in [13] the program can only be written in a subset of ANSI C.

In the second step, many types of coverage criteria [9, 20] can be used, such as *branch coverage* or *path coverage*. We use the path coverage criteria in this paper. Since there may be infinite executable paths to be covered in this criteria, we only cover the paths whose length is no more than a given value. In current implementation, we choose this value randomly, but it may also be given by analyzing the loop statements. Both BFS and DFS can be used to generate these paths. The problems of using these two procedures are discussed in [13]. What we use here is the BFS method with some restriction on the length of the program path. We also eliminate the character string function call in this step, and the details will be discussed later.

In the third step, we use a tool called EPAT, which is an extended version of the tool called PAT in [15]. It can detect the feasibility of a path and generate the test data. The tool EPAT accepts a path as input, which is generated in the second step. The path is a sequence of assignment statements and conditional expressions, preceded by a set of variable declarations. The type of the variable can be integer, real and array of them with fixed length [15]. Each conditional expression is preceded by a mark '@'.

For instance, the program segment in Example 1 is an infeasible path, which can be detected by PAT or EPAT. If we replace the first condition "`@(i>4)`" by "`@(i>=3)`", the path is feasible and we can use EPAT to detect that. What's more, we can also use EPAT to generate the test data "`i=1`" for the modified path.

Example 1 *An infeasible path which can be detected by EPAT efficiently. The mark '@' is followed by a conditional expression.*

```
int i,j;
{
    i = i + 2;
    @ ( i > 4);
    j = i;
    @ ( i + j < 8);
}
```

3.2. Character string variable

In the C programming language, a character string can be represented by a character array, or a character pointer which points to the head of a sequence of characters. Since pointers, such as pointer aliasing, can be processed in the way in our previous work [13, 14], we only focus on character array in this paper. According to the C language, an unsigned character[2] can be treated as an integer with the restriction that the value is between 0 and 255. Therefore, in the path, we can replace every character variable c by an integer variable ic with a logical expression $0 \leq ic \leq 255$. For a character array with fixed length, we replace every element of the array in that way. For the character array with unknown length, we assume the length is a small fixed value like 3, 5, or 10.

[2] In this paper, we will ignore the difference between unsigned character and character and assume every character is no less than 0.

3.3. Character string function call

In order to process function call, two methods were mentioned in our previous work [13], which were called *function modeling* and *function inlining* respectively. Both of them have their own advantages and shortcomings.

Modeling can be treated as an abstraction of the behavior. Since we want to model the behavior of a function not a big system, the formulae in first order logic (FOL) can be enough[3]. To model a function, we can analyze the function and get several constraints (may be FOL formulae) to describe its behavior. Some functions such as *getchar* are easy to model [13].

Example 2 *Let a function call be c = getchar(), if we model getchar as @* $(0 \leq INPUT \leq 255)$*, then we can translate the assignment statement into a path:*

```
{
    @ (0 <= INPUT <= 255 );
    c = INPUT;
}
```

Here INPUT is a new input variable, and the mark '@' is followed by a conditional expression.

But other functions like *strcpy* and *strcat*, which have some complicated implementations, are much more difficult to model. Another problem during modeling is how to choose the granularity of the model. We shall discuss this later.

To inline a function, one has to replace the called function by its possible paths, which means that one needs to analyze the source code of the called function. But there may be some library functions whose source code is not available. What's more, when there are loops in the function, the number of its possible paths can be infinite. Therefore, this method can not be used alone.

In this paper, we combine these two methods together to generate the test data for the character string library functions. First, we analyze the function's behavior manually, and model it using formulae in first order logic. Then, we replace these formulae by a set of sub-constraints. We call these two phases the *modeling phase* and the *expanding phase* respectively. Finally, when we generate the paths for the given program, we replace each function call by the sub-constraints. Therefore, we can eliminate the function call in the program path, and the type of the remaining variables is integer or integer array. The program path can then be solved by EPAT to generate the test data.

[3]Generally, we can not use FOL formulae to describe the behavior of any program. But, since we focus on character strings, and we also restrict the length of the string to be a constant, FOL formulae should be enough.

3.4. Modeling phase

In order to model the functions, we use the post-conditions. For each function, we use the predicate logic formulae to represent the conditions which should be satisfied after executing that function. Generally speaking, there are two kinds of functions in C. The first kind of functions do not modify any of their parameters but only their return values. Functions in the other kind modify some parameters as well as their return values.

For the functions in the first kind, such as *strlen*, *strcmp* and *strstr*, we can easily model them by FOL formulae, which represent the constraints satisfied after the called function. For example, let us consider the C string function *strlen*. It takes a character array (or a character pointer) as input, and returns an integer value which counts the useful characters (character not equal to '\0') in that array.

Example 3 *Let s be a character array with length l, then we can model the assignment statement $n = strlen(s)$ as:*

$$0 \leq n < l \wedge s[n] = 0 \wedge \forall k(0 \leq k < n \rightarrow 0 < s[k] \leq 255)$$

But, there are some difficulties when we model the functions in the second kind, such as *strcpy* and *strcat*. Because these functions modify their parameters, and we can not use FOL formulae to represent these modification. Thus we can not model these functions by FOL formulae only. One solution to this question is to add assignment statements to the model. That is, we model the functions of the second kind as FOL formulae with some assignment statements.

For example, the function *strcat*, which takes two character arrays as input, concatenates the second parameter at the end of the first one and returns the concatenated array. We can use FOL formulae and assignment statements to describe its behavior, as shown below.

Example 4 *Let s and t be two character arrays with length l, then we can model the function call $strcat(t, s)$ as:*

$$l_1 := strlen(t) \wedge l_2 := strlen(s) \wedge l_1 + l_2 < l \wedge \\ \forall k(0 \leq k < l_2 \rightarrow t[l_1 + k] := s[k])$$

Here the mark ':=' represents an assignment operation, and 'strlen' can be replaced by the model displayed in Example 3.

There is one problem when we use assignment statements in the FOL formulae. How to distinguish the value of the variables before and after the assignment operator? One straightforward way is to use two copies of variables. Use the original one and the primed one to store the value before and after the assignment respectively. But this will introduce more variables to the paths. What's more, it will make the test generation procedure much more difficult.

In this paper, we use pre-condition and post-condition. For functions in the second kind, we use both pre-condition and post-condition to describe the behavior of the called function. For example, we can use pre- and post-condition to model the function call *strcat(t,s)* as Example 5.

Example 5 *Use pre-condition and post-condition to model the function call strcat(t,s).*

- *Pre-condition:*

$$l_1 := strlen(t) \wedge l_2 := strlen(s) \wedge l_1 + l_2 < l$$

- *Post-condition:*

$$\forall k(0 \leq k < l_2 \rightarrow t[l_1 + k] := s[k])$$

Other functions which modify their parameters can also be modeled using the same method. Actually, we can use the FOL formulae and assignment statements to model most of the character string library functions.

3.5. Expanding phase

After modeling the function by FOL formulae and assignment statements, we can not use these models directly in the program path. Because there are quantifiers in the models, and the path solver EPAT can not solve the path condition containing quantifiers, we have to eliminate them. The mission of this phase is to eliminate the quantifiers from the model, and expand the model into several sub-constraints, which can be used to replace the function call in the source program. This work can also be viewed as an abstraction of *function inlining*, in which one gets the sub-constraints directly from the source code, but here we get the sub-constraints from the model. This method is easier than directly inlining and does not need the source code.

The basic idea is to eliminate the quantifiers by enumerating all the possible values. From the above examples we can see that, the quantifiers in Example 3 and Example 4 are restricted by some constraints. Then, we can enumerate all the values that satisfy these constraints. But there are other problems. Some of the constraints are related to the length of the character array, so we can not enumerate all the possible values because the length of the array may change at runtime. One method is to assume that all the character arrays have some fixed length such as 3, 5, or 10.

Based on the assumption that the character array has some fixed length, we can easily eliminate the quantifiers by enumerating all the possible values. For example, we can eliminate the quantifiers in Example 3 as shown below.

Example 6 *In Example 3, let the length of the character array s be 3, then we can eliminate the quantifiers and get the following constraint:*

$$\begin{array}{l}(n = 0 \wedge s[0] = 0) \vee \\ (n = 1 \wedge s[1] = 0 \wedge 0 < s[0] \leq 255) \vee \\ (n = 2 \wedge s[2] = 0 \wedge 0 < s[0] \leq 255 \wedge 0 < s[1] \leq 255)\end{array}$$

As you can see, the constraint in Example 6 is in Disjunctive Normal Form (DNF). This does not occur by accident. In our experiments, almost all the constraints are in DNF. Absolutely we can solve these constraints by PAT or EPAT, but sometimes the solving process can be quite slow. To remedy this problem, we can split the above constraint into several sub-constraints, use every disjunct as a sub-constraint, and then split the given program path into several sub-paths, each of which contains one of the sub-constraints. Although we get more paths than before, we can complete the whole process in less time, because each sub-path can be checked very quickly (in tens of milliseconds).

Example 7 *Eliminate the '$\vee$' operators in Example 6, and get three sub-constraints as follows:*

1. $n = 0 \wedge s[0] = 0$,
2. $n = 1 \wedge s[1] = 0 \wedge 0 < s[0] \leq 255$,
3. $n = 2 \wedge s[2] = 0 \wedge 0 < s[0] \leq 255 \wedge 0 < s[1] \leq 255$

When we generate a path that contains one call to strlen*, we can split the path into three sub-paths, each of which contains one of the above sub-constraints as a logical expression respectively .*

After we eliminate the quantifiers and get several sub-constraints, we can use one of the sub-constraints as post-condition of the function, and replace the function call statement by each of the sub-constraints as a conditional expression, which must hold after the path is executed. Therefore, we will get several paths from the original path. We call these new paths as *sub-paths*, corresponding to the original path. They also contain all the possible paths, if we assume that the character array has some fixed length.

3.6. Generate the paths and the test data

After the *modeling phase* and the *expanding phase*, we can get several disjoint sub-constraints for each string library function. Then we can generate the paths according to BFS procedure to satisfy the path coverage criteria with some restrictions on the path length. When we generate a path which contains a function call, we can split this path into several sub-paths, each of which contains one of the sub-constraints generated in the *expanding phase* to describe the behavior of the called function.

After that, we will get a set of sub-paths to satisfy the coverage criteria, and then we can use our path feasibility analysis tool EPAT to solve each sub-path. For all the sub-paths, we remove the infeasible ones and collect the test data for the feasible ones. Finally, we can get the test data for the character string variable.

4. Examples

In this section, we will use the method introduced in this paper to generate the test data for a given program.

```
char max[256];
int max_str(char *a,char *b,char *c)
{
        int     res = 0;
        int     flag = 0;
/*s1*/  strcpy(max,a);
/*s2*/  res = 0;
/*s3*/  flag = strcmp(max,b);
/*c1*/  if (flag < 0){
/*s4*/      res = 1;
/*s5*/      strcpy(max,b);
        }
/*s6*/  flag = strcmp(max,c);
/*c2*/  if (flag < 0){
/*s7*/      res = 2;
/*s8*/      strcpy(max,c);
        }
/*s9*/  return res;
}
```

Figure 1. A program returns the maximum of the given three character string parameters.

The program code is displayed in Figure 1, which is a variation of the program taken from [17]. It calculates the maximum of the given three character strings, and returns the index of the maximum. Its CFG is given by Figure 2.

After analyzing the source code, we can generate four paths for that program. Each of them contains some calls to the functions *strcmp* and *strcpy*. Using the method described in this paper, we can get the following result:

- In the *modeling phase*, we model the function *strcmp* and *strcpy* by FOL formulae and some assignment statements.
- In the *expanding phase*, based on the assumption that every character array has a fixed length of *n*, we can generate $2n$ sub-constraints for function *strcmp* and n sub-constraints for function *strcpy*. If we let *n* be 3, we can generate 6 and 3 sub-constraints for the two functions respectively.

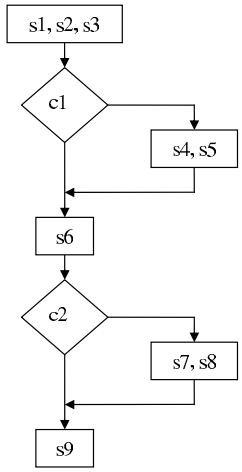

Figure 2. CFG of the program in Figure 1.

- Then we can generate the sub-paths. For the given code in Figure 1, we can generate four paths, named A, B, C and D, respectively.
  1. Path A contains the statement s1, s2, s3, c1, s6, c2 and s9. The path condition is $a \geq b \wedge a \geq c$.
  2. Path B contains s1, s2, s3, c1, s4, s5, s6, c2 and s9. The path condition is $a < b \wedge b \geq c$.
  3. Path C contains s1, s2, s3, c1, s6, c2, s7, s8 and s9. The path condition is $a \geq b \wedge a < c$.
  4. Path D contains s1, s2, s3, c1, s4, s5, s6, c2, s7, s8 and s9. The path condition is $a < b \wedge b < c$.

  Statement s1, s5 and s8 are function calls to *strcpy*, while s3 and s6 are function calls to *strcmp*. So, there are 2 calls to *strcmp* and 1 call to *strcpy* in path A, 2 calls to *strcmp* and 2 calls to *strcpy* in both path B and path C, and 2 calls to *strcmp* and 3 calls to *strcpy* in path D. After all the function calls are replaced by their corresponding sub-constraints, we generate 108, 324, 324 and 972 sub-paths for path A, B, C and D.
- After we generate all the above sub-paths, we can use our path analysis tool EPAT to solve them. The results show that, there are 14 feasible sub-paths for path A, 21 for path B, 17 for path C, and 24 for path D. That is to say, among all these 1728 paths, we have 76 feasible ones. For each of these feasible paths, we can generate the corresponding test data to execute along it. These test data cover all the paths of the program, on the assumption that each character string has no more than three elements.

Example 8 shows an example for the test data. It is generated according to one sub-path of path D. That is, it should satisfy the constraint $a < b \wedge b < c$. From Example 8, we can see that the test data for a, b and c do satisfy the path condition $a < b \wedge b < c$. If we use the test data in Example 8 to run the program in Figure 1, the program will surely execute along the path D.

Example 8 *Test data for one feasible sub-path of path D:*

```
a[0] = 1; a[1] = 0;
b[0] = 2; b[1] = 1; b[2] = 0;
c[0] = 3; c[1] = 0;
```

Although the test data are represented in integer values, we can easily translate them into some meaningful characters by adding some constraints. If we assume that every character is either 0 or between 'a' and 'z', we can get the test data as a = *"a",* b = *"ba" and* c = *"c".*

It should be noted that:

1. We do not need to generate all the sub-paths. For every path from the source code, once we find one feasible sub-path, we can return the sub-path and get the test data. That is, we only need to get one feasible sub-path for each path A, B, C or D.

2. In most programs, we may have some conditions or assertions after we find the maximum of the given strings. Then the path will contain more constraints. And there will be more sub-paths which are infeasible. Using our path analysis tool EPAT, we can efficiently find and remove these infeasible sub-paths, and generate the test data for the feasible ones. Then we will get test data for the given program very quickly.

We also use our method to model some other commonly used string-handling functions, such as *strlen*, *strcmp*, *strcpy*, *strcat*, *strstr*, *strcspn*, *strset* and so on. The bounded version such as *strncmp*, *strncpy* can also be modeled and processed using our method. All these functions are taken from the standard library and declared in the header file "*string.h*". We also use these models to generate test data for some programs which call these functions. And the results show that this method is feasible and efficient.

5. Discussion

In this section, we first discuss the granularity of the model, and then we discuss some advantages and shortcomings of our method.

5.1. Granularity of the Model

One of the most common problems during the modeling phase is how to choose the granularity of the model. We note that this choice is subtle. When the granularity is too fine, we may get in trouble by the unimportant information. On the other hand, we may lose some useful information when the granularity is too coarse.

But, this problem can be traced according to what we focus on. In this paper, because we want to describe the behavior of the function call, the important information for us is *what* the function has done but not *how* it accomplishes. So, what we focus on is its effect but not its procedure. Therefore, we use the formulae in FOL to describe the post-condition of the function call.

5.2. About our method

Through our experiments, we notice that our method has the following advantages:

1. We can generate test data for functions with complex control structures. Character string variables and many string-handling function calls can be handled.

2. Most of the string library functions taken from the header file *"string.h"* can be processed except for two: *strtok* and *strdup*, which have some complicated implementations.

3. Compared with the work done by Zhao et al. [17, 18, 19], our approach is based on static analysis, while symbolic execution and constraint solving techniques are also used. We can detect the infeasible paths more efficiently and we do not need to map a string to one big integer value.

However, our method has some limitations.

- When we replace the called function by its corresponding model, we currently use the call-by-name approach. So, the literal constant strings can not appear in the arguments of a function. This may need a better implementation.

- Function *strtok* can not be handled, because its behavior is decided by its previous behavior. But, since the GCC man page for *strtok* suggests "*It is not thread-safe*", there are few programs calling this function.

- Function *strdup* can not be processed either, because it allocates the heap memory in its implementation. But, since it is similar to the function *strcpy*, we can use the function call *strcpy* to replace it.

6. Conclusion

In this paper, a method based on static analysis and function modeling is presented to generate the test data for C programs with character string variables and some commonly used library functions. The character variable is viewed as an integer variable with the restriction that its value should be between 0 and 255, while a string variable is viewed as an array of characters. We also assume that the length of a character array is a compile-time constant. Many commonly used character string library functions taken from the header file *"string.h"* are also modeled to generate the test data for programs calling these functions. A prototype tool called StrGen is also developed to illustrate the feasibility of our method. And the results of the examples show that this method is feasible and very efficient.

In the future, we plan to extend our method such that the character pointer and memory allocation function can be handled, so that we can process any kind of character strings, not only those represented as character arrays. And we plan to do more experiments with other programs, especially those from industry.

Acknowledgement

The authors would like to thank the anonymous referees for their helpful comments. Ruilian Zhao and Sheng Liu read earlier versions of the paper and provided good suggestions.

References

[1] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the international conference on Reliable software*, pages 234–245, 1975.

[2] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.

[3] J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linköping*, pages 21–28, 1999.

[4] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA '07)*, pages 151–162, 2007.

[5] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[6] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879, 1990.

[7] N. Mansour and M. Salame. Data generation for path testing. *Software Quality Control*, 12(2):121–136, 2004.

[8] P. McMinn. Search-based software test data generation: a survey. *Softw. Test. Verif. Reliab.*, 14(2):105–156, 2004.

[9] G. J. Myers. *The Art of Software Testing, Second Edition*. John Wiley & Sons, Inc., New York, USA, 2004.

[10] M. Prasanna, S. Sivanandam, R. Venkatesan, and R. Sundarrajan. A survey on automatic test case generation. *Academic Open Internet Journal*, 15, 2005.

[11] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13)*, pages 263–272, 2005.

[12] N. Williams, B. Marre, and P. Mouy. On-the-fly generation of k-path tests for C functions. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 290–293, 2004.

[13] Z. Xu and J. Zhang. A test data generation tool for unit testing of C programs. In *Proceedings of the Sixth International Conference on Quality Software (QSIC 2006)*, pages 107–116, 2006.

[14] J. Zhang. Symbolic execution of program paths involving pointer and structure variables. In *Proceedings of the Fourth International Conference on Quality Software (QSIC 2004)*, pages 87–92, 2004.

[15] J. Zhang and X. Wang. A constraint solver and its application to path feasibility analysis. *International Journal of Software Engineering and Knowledge Engineering*, 11(2):139–156, 2001.

[16] J. Zhang, C. Xu, and X. Wang. Path-oriented test data generation using symbolic execution and constraint solving techniques. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods*, 2004.

[17] R. Zhao and M. R. Lyu. Character string predicate based automatic software test data generation. In *Proceedings of the Third International Conference on Quality Software (QSIC 2003)*, pages 255–262, 2003.

[18] R. Zhao, M. R. Lyu, and Y. Min. Domain testing based on character string predicate. In *12th Asian Test Symposium (ATS 2003)*, pages 96–101, 2003.

[19] R. Zhao and Y. Min. Automatic test data generation of character string based on predicate slice (in Chinese). *Journal Of Computer Research And Development*, 39(4):473–481, 2002.

[20] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.