International Journal of Security and Its Applications
Vol. 10, No. 6 (2016) pp.155-170
http://dx.doi.org/10.14257/ijsia.2016.10.6.16

A Target-Oriented Path Generation Method based on Concolic Testing

Zhang Xuzhou[1], Gong Yunzhan[1], Wang Yawen[1,2] and Xing Ying[1]

[1]*State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China*
[2] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*
laomao22311@126.com

Abstract

The space explosion caused by loop structure in path exploration makes it difficult to find a target-oriented path during code coverage testing. In this paper, a method named target-oriented concolic testing(TOCT) is introduced to tackle the problem of generating a path to cover a target in a loop structure. First, the static analysis method static prefix path search (SPSS) is used to generate a prefix path. Solving the constraints on the prefix path obtains a test case, with which concrete execution is carried out as an input. Meanwhile the execution trace is captured. The trace is used to unfold the loop into a feasible path. Finally SPPS is adopted again for the next prefix-path until a feasible path containing the target is achieved. Experimental results show that the proposed strategy reached a higher coverage when the targets are in loop structure and demonstrate that it is applicable in engineering.

Keywords*: automated software testing, handling loop, path generation, concolic testing*

1. Introduction

High-confidence is of vital importance to software, especially for the software in key services such as medical care, nuclear power monitoring, and aerospace engineering, *etc*. The so called confidence involves many aspects including correctness, reliability and safety. The National Security Agency of USA has been devoted to the research of high-confidence software and system (HCSS) for years [1], and tried to provide methodology of improving the quality of codes in terms of design, development and analysis, but there is much remained to be answered urgently.

As an important means to guarantee software quality, software testing plays a key role throughout the process of software development. The purpose of software testing is verifying the correctness of the function of software, and finding potential flaws as much as in the development stage. As a basic field in software testing, the white-box testing tests internal structures or workings of an application and aims at reaching high code coverage, the theory of which is a character of software flaws: the higher the probability that a program path can be executed, the lower the probability that the path contains logically erroneous and incorrect sentences [2]. Therefore, higher code coverage means lower probability that the program contains errors, and in turn the software is of higher confidence.

Automated software testing reduces the work of artificial participation on getting code coverage, thus greatly improving the efficiency. The present automated testing method can be categorized as: random testing, static testing, concolic execution.

* Corresponding Author

ISSN: 1738-9976 IJSIA

Random testing [3] generates random inputs at bare overhead; it is one of the most commonly used automated test techniques. However the random testing is not effective when program behavior needs lots of specific inputs. What's more, most of program behaviors are triggered only by a small range in comparison with input universe. Static testing is generally based on symbolic execution [4-6, 12]; it treats every input value as a symbol and gathers symbolic constraints along an execution path. The goal is then generating concrete inputs that satisfy the constraints. But static analysis cannot be so precise that may give too many false positives.

In recently years, concolic testing [7-9] has been proposed as a combination of symbolic execution and concrete execution. The program is executed with some given inputs, while simultaneously performing concrete execution to get an execution path and symbolic execution to collect symbolic constraints which have been simplified by concrete values. Then with a negative on a selected condition, symbolic constraints are then used to generate a new path with the same prefix. The concolic testing incrementally explores paths to achieve code coverage.

In practice, however, for both static testing and concolic testing, when there is loop structure in program under test (PUT), the possible number of paths is so large that all the methods end up with exploring only small parts of program state space. For example, with just a few loops of moderate numbers of iteration, the complete coverage becomes impractical because the combinatorial explosion when we concatenate nested loops, meanwhile the feasible paths are relatively few. Consequently, we must deal with loops besides the above techniques. The static testing often use 0-1 strategy, that treats loop as a conditional statement with no iteration, or fixes the iteration of loops with a constant number [10]. Concolic testing tools such as DART [7], CUTE [8] try to explore program state space and set some flags and thresholds to avoid infinite iteration. [11] proposed a fitness function to guide path exploration.

We proposed target-oriented concolic testing (TOCT) to tackle the problem of generating path to cover a target in a loop structure. The framework consists of static prefix-path search (SPPS) and concrete execution (CE). TOCT performs SPPS and CE alternately to incrementally generate a feasible path.

Based on Code Testing System (CTS) (http://ctstesting.com), we made the following contributions in this paper.

(1) The search scale of generating paths for loop structure is discussed. And it is reduced greatly by changing the target of path generating.

(2) We introduced TOCT framework which combined static analysis and actual execution to solve the problem of covering the target in a loop structure.

In this paper, statistical analysis is made on the performance of the proposed framework, and comparison is made with the methods using other loop-handling strategies in terms of effectiveness. The experimental results demonstrate the advantage of the proposed method in processing the target element in a loop structure.

The rest of this paper is organized as follows. In Section 2, the problem of path generating in loop structure is defined and the scale of the search space is analyzed. The framework combining static analysis and actual execution is proposed in Section 3. In Section 4, a case study is utilized to thoroughly explain the implementation of the proposed method. In Section 5, Empirical experiments are made and the results are statistically analyzed to show the effectiveness of our method. This paper is concluded and the directions of our future research are highlighted in Section 6.

2. Reformulation of Target Oriented Path Generation

This section provides definitions for target-oriented paths generation, and introduces the scale of path search space.

2.1. Definition of Path Generation Problem

In white-box testing, control flow graph (CFG) is often used to represent the control flow structure of the PUT. It is a directed graph and normally denoted as CFG=(N, E, i, o), where N is a set of nodes, E is a set of edges, and i and o are respective unique entry and exit nodes to the graph. Each node n∈N is a statement in the program, and each edge (T(e), H(e))∈E is the connection between a pair of adjacent nodes with T(e) representing the tail of e and H(e) representing the head of e.

When it comes to the target t to be covered, it is different with different coverage criteria. Statement coverage is used as the example in the following explanation, and in this case t representing target element is a statement node in the CFG. A t-oriented path generation is to generate a sequence Path=n1, n2,...,nlength , where n1=I, nlength =o, and t∈Path.

To generate paths on a directed graph is in fact the problem of traversing a graph, which is solved by search methods. In our implementation, the search process is modeled as state space search, and the definition of state space follows.

Definition 1. The *state space* is a quadruple (*S*, *A*, *I*, *F*), where *S* is a set of states, *A* is a set of arcs or connections between the states that correspond to the steps or operations of the search at different states, *I* is a non-empty subset of *S* denoting the initial state of the problem, and *F* is a non-empty subset of *S* denoting the final state of the problem.

Definition 2. A *state* is a quadruple *s*= (*Node*, *Successor*, *Path*). *Node* is the current node; Successor provides a link to the sets of the successive nodes of the current node; *Path* is the sequence of nodes that have been traversed; When Node is the first node of the CFG, s is the initial state. While when *Successor* is *empty* and *t*∈*Path*, s is the final state. *Path* in the final state is one solution to the problem.

2.2. Scale of Search Space

The above mentioned state space E can be represented as a search tree. In a search tree for loop structures, nodes are divided into three categories: end node, branch node, statement node:

End node: or leaf node, marks the end of loop iteration, the degree of which is 0; icon is overlapped circles

Branch node: the entrance of each loop. Its child nodes represent different branches of one iteration with degree of *m*+1, consist of 1 end node and m statement nodes

Statement node: represents a statement on a branch of a single loop iteration, the degree of which is 1.Target *t* is a statement denoted as a blue circle. A statement without t is denoted as a gray circle.

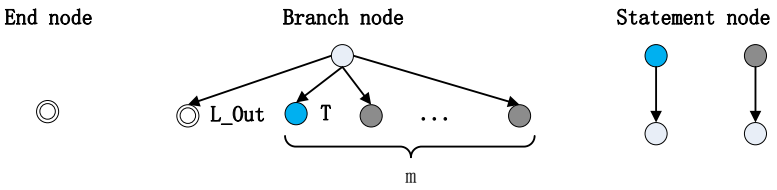

Figure 1. The Denotation of Three Kinds of Nodes

Figure 2 shows an example of a loop structure, and its corresponding CFG and search tree.

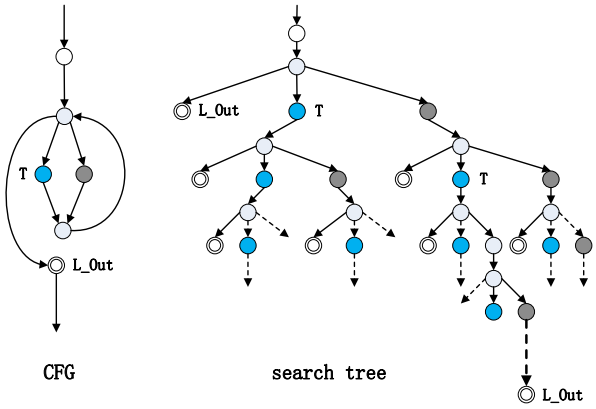

Figure 2. A Loop Examples

To the search tree, the destination path is a sequence of nodes traversed that starts from the root and ends at a leaf node. The complexity of traversing a tree is O (num), where num is the number of the leaves in the tree, so the number of nodes in a search tree directly influence the complexity of path generation.

Assume that a loop is iterated for n times, there are m branches in the loop with the number of nodes of each being a constant, and the target t is inside one of the branches.

In this example, for the search tree that corresponds to the loop structure, the depth of the root node is 0 and height is 2n. If nodes (i) is the number of nodes in the ith level, we have the following formulae.

$$nodes(0) = nodes(1) = 1;$$
$$nodes(i) = \begin{cases} (m+1) \times nodes(i-1), if\ \ i = 2j,\ j \in [1, n-1] \\ nodes(i-1) - 1, \quad if\ \ i = 2j+1,\ j \in [1, n-1] \\ nodes(i-1), \quad if\ \ i = 2n \end{cases}$$

(1)

Since a statement node has only one child, the levels of 2j and 2j+1 can be merged into one level, which is looked as the jth level of the tree, then the number of nodes is:

$$nodes(i) = \begin{cases} 2, & i = 0 \\ m^{i-1} + 2 \times m^{i}, & i \in [2, n-1] \\ m^{n-1}, & i = n \end{cases}$$

(2)

Then the total number of nodes in the search tree is calculated out as:

$$\sum_{i=0}^{n} nodes(i) = 2 + \sum_{i=1}^{n-1} \left(m^{i-1} + 2 \times m^{i} \right) + m^{n-1} = 2 + m^{0} + \sum_{i=1}^{n-1} \left(m^{i} + 2 \times m^{i} \right) = 3 + 3 \sum_{i=1}^{n-1} m^{i}$$

(3)

It can be seen that the search scale increases exponentially with the numbers of both branches and nests. Since the process of state space search is the process of generating paths, we propose a method combining static analysis and concrete execution to search the state space, which will be discussed in the following section.

3. Target-Oriented Concolic Testing

This section proposes TOCT, a framework combining static analysis and concrete execution to automate generate target-oriented path of loop structure.

3.1. The Framework of TOCT

The main idea of this paper is to incrementally generate the path in a concolic manner. First, SPPS are adopted to generate the path before the loop structure or make the path cover the target. Then CE is utilized to unfold the loop or obtain the complete path covering the target.

Table 1. Some Methods and their Description used in this Paper

Name	*Description*
Target Oriented Concolic Testing (TOCT)	*The concolic framework of path generation*
Static Prefix-path search (SPPS)	*To generate the prefix path using static analysis*
Concrete execution (CE)	*To get the successive path by concrete execution*

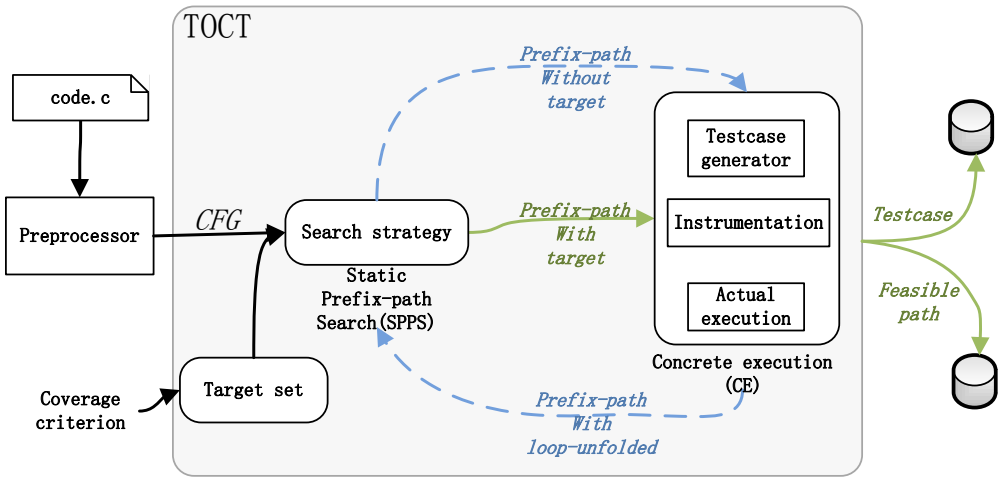

Figure 3. Overview of TOCT

The processing framework is shown by Figure 3. In the left part, the preprocessing module preprocesses the .c file of the source file, and generates the corresponding CFG. The input of the processing framework is the CFG and a set of elements to be covered, which is different with different coverage criteria. The concolic path generation method is adopted to handle each element in the set with the following steps. First SPPS is used to generate the prefix path, with two possible cases. One is that there is no target in the loop, and the prefix is part of the path from the entrance of the PUT to the entrance of the loop. The other is that the target is in the loop, and the prefix is part of the path from the entrance of the PUT to the target. Then CE (concrete execution) is conducted using the prefix path as input. The instrumentation information from CE will be analyzed to obtain the execution trace. If the target is not in the execution trace, then the path containing the loop after the prefix is cut out from the execution trace, and used as input for PPS to incrementally generate the path. If the target is in the execution trace, meaning that the feasible path covering the target is obtained, then the path is added to the set of feasible paths and the target on the path is deleted from the set of elements. Test cases are generated for the path and added to the set of test cases. The above procedure is repeated

for the uncovered elements in the set until all the elements are covered. The above mentioned process can be described by the following algorithm.

Algorithm1. Target Oriented Concolic Testing
Input *CFG g ,set of coverage targets Targets*
Output *set of feasible path Fpaths and test cases TCs*
Begin
1: ***while Targets*** $\neq \emptyset$ ***do***
2: *target* $\leftarrow$*select(Targets);*
3: *initial state*$\leftarrow$*(in, Successor*$_{in}$ *, <in>);*
4: $S_{cur} \leftarrow$*initial state*
5: ***while*** $S_{cur} \neq$ *null* ***do***
6: $Ppath_{cur} \leftarrow$ *call* ***Algorithm2 Static path Search***(S_{cur})*;*
7: $Path_{trace} \leftarrow$*call* ***Algorithm3 Concrete execution***$(Ppath_{cur})$*;*
8: ***if*** *target* $\in Path_{trace}$ ***then***
9: *Fpaths* $\leftarrow$*Fpaths* $\cup Path_{trace}$*;*
10: *remove coverd targets from Targets;*
11: ***break****;*
12: ***endif***
13: ***if*** $Path_{trace} \neq$ *null* ***then***
14: $Node_{cur} \leftarrow$*getLastNode(*$Path_{trace}$*);*
15: $Successor_{cur} \leftarrow$*getSuccessor (*$Node_{cur}$*);*
16: $S_{cur} \leftarrow$ *(*$Node_{cur}$ *,* $Successor_{cur}$ *,* $Path_{trace}$*);*
17: ***endif***
18: ***endwhile***
19: *testcase* $\leftarrow$*CaseGenerator (*$Path_{trace}$*);*
20: *TCs* $\leftarrow$ *TCs* $\cup$*testcase;*
21: ***endwhile***
22: ***return*** *Fpaths, TCs;*
End

3.2. Perfix-Path Generation by SPPS

According to Definition 1, the generated path should be a complete path, and according to the discussed search scale in Section 2.2, when the path has come to the target, it is still required to traverse to the exit node so as to reach the final state, causing a big cost at searching the path after the target. Intuitively, the path scale after the target is far bigger than that before it.

The scale of the search space is still discussed as the form of Section 2.2, when the termination condition is changed to the target element, after the path to the destination need not be considered, then the prefix of the path is the first target point t necessarily occur (if it is the first time they should have been terminated before the search), so at this time of the search tree node has changed

End node: or leaf node, marks the target statement, the degree of which is 0, and is denoted as the double blue center circle.

Branch node: the entrance of each iteration. Its child nodes represent different branches of one iteration with degree of m, consist of 1 end node and m-1 statement nodes

Statement node :represents a statement without target on a branch of a single loop iteration, the degree of which is 1.

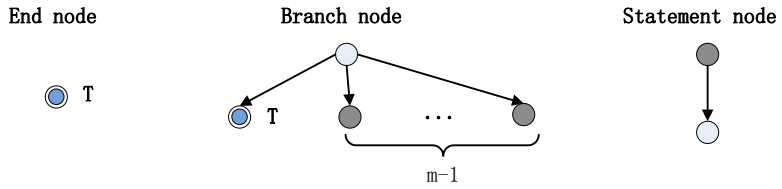

Figure 4. The New Denotation of Three Kinds of Nodes

Then the number of nodes is:

$$nodes(i)=\begin{cases}2, & i=0\\ (m-1)^{i-1}+2\times(m-1)^{i}, & i\in[2,n-1]\\ (m-1)^{n-1}, & i=n\end{cases} \quad (4)$$

The total number of the nodes in the search tree can be calculated as

$$\sum_{i=0}^{n} nodes(i)=2+\sum_{i=1}^{n-1}\left((m-1)^{i-1}+2\times(m-1)^{i}\right)+(m-1)^{n-1}=3+3\sum_{i=1}^{n-1}(m-1)^{i} \quad (5)$$

Compared with formula (2), the base of the geometric progression is decreased by 1.

The above calculations show that the search space can be reduced by appropriately changes the termination condition, the same effect as applying a pruning strategy. When the loop containing the target element t, it will be able to get a path reach t in a limited iteration times of unfolding loops. So the search termination condition is unfolding the loop until reach t .When the loop does not contain t, it cannot select a certain paths which can for sure reach t. So when the loop does not contain a target element, the search termination condition is the head node of loop structure. Following is the definition of prefix-path.

Definition 3. A prefix-path Ppath of target t is a Path=n1, n2,...,nlength, where n1=i, nlength =t or the head node of a loop (without target).

This method can reduce the scale of the search space. When the search strategy keeps unchanged, the scale of the generated complete path is larger than the scale of the generated prefix path. In the implementation of this paper, depth-first-search is used to faster reach the loop structure containing the target. While when generating path for the loop structure, the purpose is to obtain the prefix path in a finite-time loop unfolding, and depth-first-search (DFS) will cause the unlimited extension of a single branch, so a second round of iteration is carried out if all the paths are searched after a round of iteration. If all the paths in a round of iteration is regarded independent of each other, then breadth-first-search (BFS) can be used for the search inside the loop. To avoid unlimited iterations, a upper limit MAX_ITRAT is set to restrain the rounds of loop iterations. DFS and BFS are implemented by a queue and a stack, respectively. The above mentioned search process is described by the following algorithm.

Algorithm2. Static Prefix-path search
Input *state S*
Output *prefix_path Ppath*
Begin
1: ***if*** *node(S)=in* ***then***
2: *stack.push(S);*
3: *loop_count ←0;*

```
4:   while loop_count < MAX_ITRAT && stack is not empty do
5:       loop_count++;
6:       S_cur ←stack.Top();
7:       Nodecur ← Node(S_cur);
8:       Successorcur ← Successor(Scur);
9:       if accept(S_cur) then
10:        return Path(Scur);
11:      while Successorcur hasnext do
12:          Nodenext ← Successorcur.getnext();
13:          Snext ← (Nodenext, Successor Nodenext。 ,  Path(Scur) ∪Nodenext);
14:          if Nodenext ≠ loop_out then
15:            stack.push(S_next);
16:            Ppath ←call Prefix-path search (S_next);
17:            if accept(Ppath) then
18:              return Ppath;
19:            stack.pop();
20:            loop_count--;
21:            if stack is empty then
22:              stack.push(queue.dequeue());
23:              queue.pop();
24:          else
25:             stack.push(S_next);
26:             queue.inqueue(S_next);
27:      end while
28:  end while
End
```

3.3. Successive Path Generation by CE

Following the SPPS, it is required to generate the successive path inside the loop structure. Inspired by concolic testing, in this paper the successive path is obtained by capturing the trace of the concrete execution.

Concolic testing executes a test case while in parallel collecting symbolic constraints along the actual execution path, then select a condition to negate the constraint and solve the new constraints to get the next execution path. The two paths share a same prefix path which divaricates after the selected condition and takes different branch. That means if solving the constraints of a prefix path to get a test case, the concrete execution to this test case must lead to a feasible path started with the prefix path, then takes an alternative branch to the end. The details can be described by the following proposition.

Proposition. If there is a test case satisfies the constraints prefix paths, tracks the actual implementation of this test is to prefix the path up to the beginning of this path.

Proof. Let D^0 be the initial input domains of all input variables, after a prefix-path Ppath's constraints would be D^{Ppath}, $D^{Ppath} \subset D^0$, Ppath is infeasible iff $D^{Ppath} = \varnothing$

Assuming that there are m feasible paths which started with Ppath , let Ppath* be the set of these paths , each $Ppath^*_i (1 < i < m)$ has a corresponding domains D^{Ppath_i}, then $D^{Ppath_i} (1 < i < m)$ are partition of $D^{Ppath} \subset D^0$ partition

$$\forall i, j \in [1, m], i \neq j \Rightarrow D^{Ppath_i} \cap D^{Ppath_j} = \varnothing and \bigcup_{i=1}^{n} D^{Ppath_i} = D^{Ppath}$$

Then a value is selected from D^{Ppath} that must belong to a D^{Ppath_i}, which can be mapped to a corresponding feasible path $Ppath^*_i$.

We can make use of this property to obtain the successive path for unfolding loop. At first use a constraint solver to generate test case of prefix path Ppath, carry out concrete execution with test case as an input while obtain the execution trace at the same time. If trace contains target, the trace is a feasible path of target. If the trace does not contain the target but it starts with Ppath, then it is used to unfold successive loop structure. When fail to generate test case indicate that the Ppath is not prefix of a feasible path. The above mentioned search process is described by the following algorithm.

Algorithm3. Concrete execution
Input *prefix-path Ppath*
Output *Prefix-path with loop-unfolded* ***or*** *Feasible Path* $Path_{trace}$*;*
Begin
1: $TC_{ppath} \leftarrow CaseGenerator(Ppath);$
2: $Trace \leftarrow ExecuteTestCase(TC_{ppath});$
3: ***if*** $target \in Trace$ ***then***
4: $Path_{trace} \leftarrow Trace;$
5: ***else if*** $Ppath \in Path_{trace}$ ***then***
6: $Path_{trace} \leftarrow getLoopTraceAfterPpath(Trace);$
7: ***else***
8: $Path_{trace} \leftarrow null;$
9: ***return*** $Path_{trace};$
End

4. Case Study

To better demonstrate the effectiveness of our strategy, the example in Figure 5 is used. The input of this program segment is an integer array with length n, and the function is to judge whether the largest number in a[n] is a prime number. There are two loop structures. The first loop is to obtain the largest number, and the second is to judge whether it is a prime number. To make a thorough explanation, we choose a break statement s10 in the second loop as the target.

```
Example：
.void arrayPrime(int a[],int n){
s1          int i,k=0,prime=1;
            int max = a[0];
s2.         if(n<0)
s3.           return ;
s4.         for(i=1;i<n;i++){
s5.           if(a[i] > max)
s6.             max = a[i];
            }
s7.         k=sqrt(max+1);
s8.         for(i=2;i<=k;i++) {
s9.           if(max%i==0){
                prime=0;
s10           break;//cover target
              }
            }
s11.        if(prime) {
s12.          printf("%-4d is a prime \n",max);
              }
s13.        return;
```

Figure 5. A Example of PUT

The process of path generation is shown by Tables 2 and 3, with steps denoting the sequence that PPS and CE occur. First the initial state is constructed as the input for PPS. At state st_3, the current node s3 is a return statement, its successor is null, Path is not a

prefix path, and accept(st_3) returns false, so the top state st_3 is popped out and the new state st_4 is pushed into the stack, as shown in bold. Path of st_4 is a prefix path, accept(st_4) returns true, Path=<s1,s2,s4> is output as the Prefix-path of st_4, and the PPS step of the first stage ends. The input of CE in step 2 is the output of step 1, which is Prefix-path=<s1,s2,s4>, and the obtained path constraint is $\neg(n<0)$. Constraint solving returns n=3, and there is no constraint in a[], so a group of values {4,7,2,10,5,10} with random length is generated. The concrete execution with this group of values and the analysis on the instrumentation information leads to the execution trace Trace=<s1,s2,s4,s5,s6,s4,s5,s4,s7,s8,s9,s8,s11,s13>, which starts from a prefix path and does contain the target s10, so the path after the prefix path in the execution trace which is Pathtrace =<s1,s2,s4,s5,s6,s4,s5,s4,s7> is the output of CE in the second stage. In step 3, first a state is constructed with the output Pathtrace of CE in step 2, with Node being the tail node of $Path_{trace}$, Successor being the successive node of Node, and Path being $Path_{trace}$. The newly constructed state is the input of PPS. At state st_6, Node=s8 is the entrance node to the loop structure, the target is not in the loop, so Path is not a prefix path, and accept(st_6) returns false, as shown in bold. At state st_8, s8 is selected from the set of the successors of s9 ({s10,s8}), meaning that a round of iteration is completed and the entrance node of the loop is reached again. Since the first round of iteration does not search out all the paths, the state st_8 is inserted into Queue, as shown in bold. The traversal to the top state st_7 goes to state st_9. Here $Node_{cur}$ is the target, Path is a prefix path, and accept(st_8) returns true. Path=<s1,s2,s4,s5,s6,s4,s5,s4,s7,s8,s9,s10> of st_8 is output as Prefix-path, and PPS of Step3 ends. The input of CE in step 4is the output of step 3, which is Prefix-path, solving the constraints obtains the test case, which is then executed. The target is inside Trace, which is a feasible path as the output of step 4. The path generation process aiming at the target s10 succeeds.

Table 2. Process of SPPS

Step	$State_{cur}$: *{Node, Successor, Path}*	State_stack [Top,...,Bottom]	Queue [Head,...,Tail]	Accept($State_{cur}$)
1	Input: {s1,{s2},<s1> }			
	st_1={s1,{s2},<s1> }	[st_1]	--	False
	st_2={s2,{s3,s4},<s1,s2> }	[st_2,st_1]	--	False
	st_3={s3,**null**,<s1,s2,s3> }	[**st_3**,st_2,st_1]	--	False
	st_4={s4,{s5,s7},<s1,s2,s4> }	[**st_4,**st_2,st_1]	--	True
	Output :*Prefix-path*=<s1,s2,s4>			
3	Input: { s7,{s8},< s1,s2,s4,s5,s6,s4,s5,s4,s7>}			
	st_5={s7,{s8}, < s1,s2,s4,s5,s6,s4,s5,s4,s7>}	[st_5,st_4,st_3,st_2,st_1]	--	False
	st_6={s8,{s9,s11}, < s1,s2,s4,s5,s6,s4,s5,s4,s7,s8>}	[st_6,st_5,st_4,st_3,st_2,st_1]	--	**False**
	st_7={s9,{s10,s8}, < s1,s2,s4,s5,s6,s4,s5,s4,s7,s8,s9>}	[st_7,st_6,st_5,st_4,st_3,st_2,st_1]	--	False
	st_8={s8,{ s9,s11}, < s1,s2,s4,s5,s6,s4,s5,s4,s7,s8,s9,s8>}	[st_7,st_6,st_5,st_4,st_3,st_2,st_1]	**[st_8]**	False
	st_9={s10,{s8}, < s1,s2,s4,s5,s6,s4,s5,s4,s7,s8,s9,s10>}	[st_9,st_7,st_6,st_5,st_4,st_3,st_2,st_1]	[st_8]	True
	Output :*Prefix-path*=< s1,s2,s4,s5,s6,s4,s5,s4,s7,s8,s9,s10>			

Table 3. The Process of CE

Step	Path constraints	Test case	Trace	Trace result
2	Input: *Prefix-path*=<s1,s2,s4>			
	$\neg(n<0)$	n:3 a[]:{4,7,2,10,5,10}	<s1,s2,s4,s5,s6,s4,s5,s4,s7,s8,s9,s8,s11,s13>	$Ppath \in Path_{trace}$
	Output : $Path_{trace}$ =< s1,s2,s4,s5,s6,s4,s5,s4,s7>			

4	Input: *Prefix-path*=< s1,s2,s4,s5,s6,s4,s5,s4,s7,s8,s9,s10>			
	$\neg(n<0)\wedge n>1\wedge a[1]>a[0]\wedge$ $n>2\wedge\neg(a[2]>a[1])\wedge\neg(n>3)\wedge$ $sqrt(a[2]+1)>=2\wedge a[2]\%2=0$	n:3 a[]:{0,10,4,2,4}	< s1,s2,s4,s5,s6,s4,s5,s4,s7,s8,s9,s10,s11,s13>	*target* $\in Path_{trace}$
	Output : $Path_{trace}$ =< s1,s2,s4,s5,s6,s4,s5,s4,s7,s8,s9,s10,s11,s13>			

5. Experimental Analyses and Empirical Evaluations

To observe the effectiveness of SPPS, we carried out a large number of experiments. In Section 6.1, experiments were conducted to make a comparison on coverage and execution time with two other path generation strategies. In Section 6.2 experiments were conducted to evaluate the performance when applying to different nested loops or concatenated loops. In Section 6.3, some programs from projects in engineering were used to test the ability of TOCT.

5.1. Comparison Experiments

In this section, Comparison is made between our strategy and two different strategies mentioned above: 0-1 and exhaustive search. The 0-1 strategy restricts the iteration of loop to 0 or only 1. That is to say, it regards loop condition as a condition branch with no back edge. The exhaustive strategy, on the other hands, does not restrict loop number, and continues extending the loop until the path that contains the target element is found. A threshold of execution time is used to avoid infinite circulation when exhausting loops. If the execution time is longer than the threshold, the target will be abandoned. The three strategies are all implemented in CTS, which are the same in terms of test case generation strategy, execution time statistical method and coverage statistical method, except path generating strategy.

Adopting statement coverage, the functions used include five fundamental functions (sum array, prime factorization, bubble sort, figures output), and six functions which contain loops from an engineering project aa200c. Table 4 shows the features of the functions. “Input-dependent loops” indicates whether the execution path in loop is relevant to the input variables.

Table 4. Function Features

Function	Number of loops	Input-dependent loops	Max depth of nested loops	Max length of concatenation loops
angles-angles	2	1	1	2
annuab-annuab	3	3	1	3
constel-showcname	3	2	2	2
division	4	4	2	1
prime	1	1	1	1
rplanet-reduce	4	3	1	4
s2	3	3	2	1
bubble sort	2	2	2	1
star	3	3	2	1
fk4fk5_fk4fk5	6	4	2	4
refrac_refrac	1	0	1	1

We conducted each test for 100 times, and calculated out the time consumption and the statement coverage of each test. The average data are listed below. The data of the best performance are in bold in Table 5. It can be seen that our strategy performed the on two

functions (angles ,annuab)with the highest statement coverage and the least execution time. The 0-1stretegy gained the least execution time on nine functions, but the statement coverage of four functions (constel-showcname, rplanet-reduce, fk4fk5, refrac) is less than 40%, but we can get a higher coverage to 82.2%. The achieved coverage of the other five functions (division, prime, s2, bubble sort, star) ranged from 60% to 90% using 0-1,while our strategy achieve 100% for three of them.

From the average statement coverage shown in Table 5, it can be seen that our strategy did well in five functions; the exhaustive strategy worked well in two functions (rplanet-reduce, s2)with 100% coverage, but it time consumption is 3 times of ours. The 0-1 strategy reached 0% coverage with three functions (angles, annuab, fk4fk5), because there are input-independent loops in the functions. It is impossible to generate paths for a loop with more than one iteration when applying 0-1stretgy.

Table 5 .The Comparison Result on Execution Time and Coverage

Function	Average time of execution(s)			Average statement coverage		
	0-1	**exhaustive**	**current**	**0-1**	**exhaustive**	**current**
angles	**34.0**	**345.1**	**28.2**	0.0%	93.2%	**95.6%**
annuab	52.2	955.5	**45.5**	0.0%	89.4%	**100.0%**
constel-showcname	**1349.8**	2699.9	1754.8	13.0%	42.0%	**66.7%**
division	**6.0**	35.1	15.6	87.2%	100.0%	100.0%
prime	**9.9**	58.9	10.0	85.1%	100.0%	100.0%
rplanet-reduce	**75.3**	903.4	301.2	23.0%	**92.4%**	87.0%
s2	**31.1**	140.1	43.5	64.3%	**100.0%**	89.4%
bubble sort	**16.7**	265.4	26.4	75.1%	100.0%	100.0%
star	**2.7**	2.8	8.7	100.0%	100.0%	100.0%
fk4fk5	**32.6**	27942.3	1247.9	0.0%	68.9%	**86.2%**
refrac	**78.3**	496.2	117.3	36.0%	36.0%	**89.0%**

5.2. Perform Evaluation

This part carried out experiments to evaluate the performance of TOCT for varying depth of nested loops and length of concatenated loops. This was accomplished by repeatedly running TOCT on test programs having depth (or length) of nested(or concatenated) loops varying from 1 to 10. Adopting statement coverage, every loop structure is input-dependent loop with an **if** condition statement in loop.

The experiments for depth and length were each tested 100 times, and the execution time and coverage for each test were recorded except depth of 10 and length of 10 as the execution time of a single experiment time is longer than two hours. The results can be seen in Figure 6 and Figure 7.

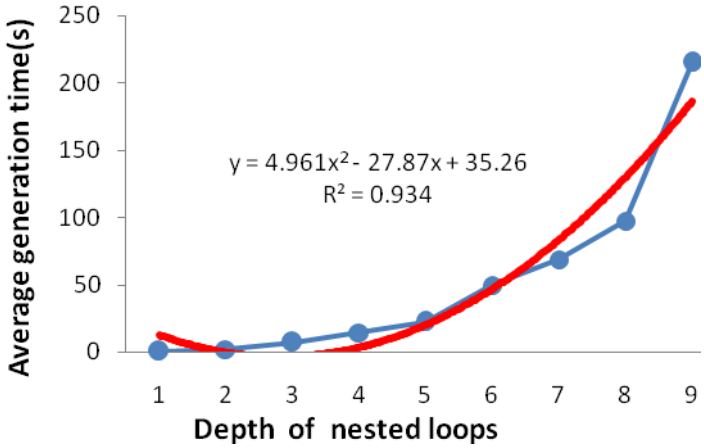

Figure 6. The Execution Time of TOCT for Different Depth of Nested Loops

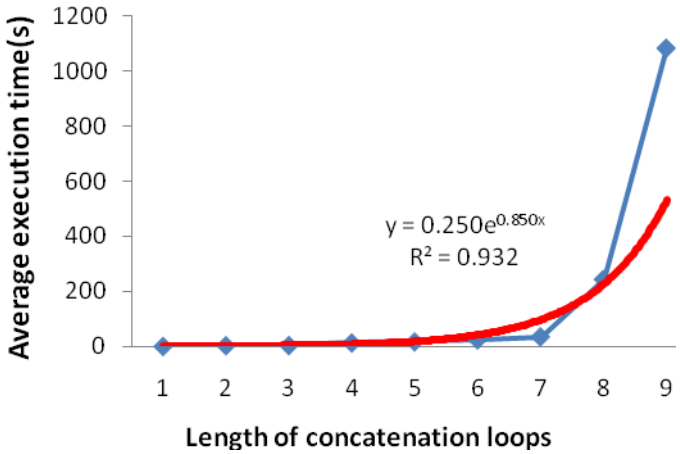

Figure 7. The Execution Time of TOCT for Different Length of Concatenated Loops

The result is shown in Figure 6 and Figure 7. Figure 6 is the execution time for different depth of nested loops. The result had been fitted into quadratic function curve, the derivative of which is 0 when x =2.8, *i.e.*, when depth of nested loops is larger than 3, the execution time starts to grow. Figure 6 is the execution time for different length of concatenated loops. The fitted curve is an exponential function; the execution time is within 100s when length of concatenated loops is less than 6. The slope becomes large when the length is more than 6, which means the execution time begins to increase. That is because with the increment of the number of concatenated loops and nested loops, the path will scale exponentially, which is consistent with the scale of search space calculated in Section 3.2.

When comparing the execution time between concatenated loops and nested loops, the time of nested loops is shorter than that of concatenated loops with the same depth /length. For the same target in the same depth / length (for example, in the innermost loop of a 4-depth nested loop /in the last loop of 4 concatenated loops), the target in nested loops will be reached during SPPS at the **first** step of TOCT because the outer loops contain the target. But in the concatenated loops, the loop before the last loop must be unfolded before selecting prefix path for the target, every unfolding requires an iteration of SPPS&CE, so the execution time of concatenated loops is longer than nested loops.

The result shows that when depth of nested loop is below 4 and length of concatenated loop is below 6, the execution time is shorter than 30s. Some programming specification [13] states that a maximum nesting depth of five levels is considered tolerable. One of McCabe's original applications [14] was to limit the Cyclomatic complexity of routines during program development; he recommended that programmers should count the complexity of the modules they are developing, and split them into smaller modules whenever the cyclomatic complexity of the module exceeded 10. In our experiment, each loop contains a set of if condition statement and the true and false branches, which means that the cyclomatic complexity of each loop is 3. When the length of concatenated loop is over 4, cyclomatic complexity is over 12 which exceeds the recommended cyclomatic complexity. So the strategy is significative under this prerequisite.

5.3. Testing Projects in Engineering

To evaluate the capability of TOCT, we carried out experiments using some programs from the project aa200c(http://www.moshier.net/) as the test beds. The results are shown in Table 6. Experimental data show that TOCT can reach more than an average coverage higher than 90%. Since the test beds were all from real-world programs, the experimental results demonstrate the applicability of the proposed method in engineering.

Table 6.The Result of Testing the Projects in Engineering

File	Function	LOC	Coverage criteria			File	Function	LOC	Coverage criteria			File	Function	LOC	Coverage criteria		
			Statement	Branch	MC/DC				Statement	Branch	MC/DC				Statement	Branch	MC/DC
dms.c	deltap	53	71%	71%	69%	refrac.c	refrac	34	100%	100%	97%	rotate.c	midentity	12	100%	100%	97%
dms.c	showcor	14	100%	100%	100%	vearth.c	velearth	33	100%	100%	100%	fk4fk5.c	fk4fk5	60	80%	85%	84%
dms.c	mod360	13	83%	85%	94%	angles.c	angles	31	100%	100%	100%	monjpl.c	moonll	59	100%	100%	97%
dms.c	showrd	28	100%	100%	96%	nutate.c	sscc	21	100%	100%	100%	monjpl.c	domoon	86	100%	97%	97%
kfiles.c	fincat	36	60%	64%	80%	nutate.c	nutlo	77	95%	97%	97%	sun.c	dosun	67	100%	100%	97%
kfiles.c	getstar	73	68%	50%	45%	nutate.c	nutate	31	100%	100%	100%	opram.c	oparams	115	100%	100%	97%
lightt.c	lightt	52	100%	100%	100%	preces.c	precess	115	100%	100%	100%	annuab.c	annuab	32	100%	100%	97%
rplanet.c	reduce	61	100%	100%	100%	constel.c	whatconstel	29	100%	100%	97%	rstar.c	rstar	71	95%	96%	95%
kepj.c	kepler	16	100%	100%	100%	constel.c	skipwh	8	75%	66%	20%	trnsit.c	search_halve	32	100%	100%	84%
epsiln.c	epsiln	39	100%	100%	100%	lonlat.c	lonlat	40	100%	100%	97%	trnsit.c	no_rise_set	105	92%	91%	89%
trnsit.c	iter_trnsit	150	84%	86%	86%	trnsit.c	trnsit	83	95%	96%	95%	**Average**		52.38	93.69%	93.25%	90.84%

6. Conclusion

As an important means to guarantee software quality, automated software testing plays a key role throughout the process of software development. When there are loop structures in the PUT, it is almost impossible to find a path throughout the program state space. We proposed target-oriented concolic testing (TOCT) to tackle the problem of generating path to cover a target in a loop structure. The framework consists of static prefix-path search (SPPS) and concrete execution (CE). TOCT performs SPPS and CE alternately to incrementally generate a feasible path. The results of comparison experiments show that TOCT obtain a higher coverage than current loop handling strategy. The experiment of testing programs from real-world project demonstrates the applicability of our method.

Our future research aims at increasing efficiency of the sequential execution of SPPS and CE. We will also focus on the performance problem exposed during the experiment

of performance evaluation, when the depth\length of nested\concatenated loops become large, the execution time is intolerable. Some parameters of the framework will be studied to achieve a trade-off between effectiveness and efficiency.

Acknowledgements

This work was supported by the National Grand Fundamental Research 863 Program of China (No. 2012AA011201), the National Natural Science Foundation of China (No. 61202080), the Major Program of the National Natural Science Foundation of China (No. 91318301), and the Open Funding of State Key Laboratory of Computer Architecture (No. CARCH201201).

References

[1] NSA. The Next Wave: The national security agency's review of emerging technologies, vol. 19, no. 1, **(2011)** [EB/OL]. http://www.nsa.gov/research/tnw/tnw191/articles/pdfs/TNW_19_1_Web.pdf.
[2] Pressman R S. Software Engineering: A Practitioner's Approach, **(1996)** [J]. ISBN: 0-07-052182-4.
[3] C. Pacheco, S. K. Lahiri, M. D. Ernst and T. Ball, "Feedbackdirected random test generation", In Proc. ICSE, **(2007)**, pp. 75–84.
[4] J. C. King, Symbolic Execution and Program Testing.Communications of the ACM, vol. 19, no. 7, **(1976)**, pp. 385–394.
[5] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala and R. Majumdar, "Generating Test from Counterexamples", In ICSE 04: International Conference on Software Engineering, IEEE, **(2004)**, pp. 326–335.
[6] Z. Xu, J. Zhang and Sim C, "Automatic Test Data Generation for Unit C Programs [C]", In: proceedings of the Sixth International Conference on Quality Software (QSIC **2006**).
[7] P. Godefroid, N. Klarlund and K. Sen, "DART: Directed automated random testing", InPLDI 05: Programming Language Design and Implementation, ACM, **(2005)**, pp. 213–223.
[8] K. Sen, D. Marinov and G. Agha, "CUTE: A concolic unit testing engine for C", In FSE 05: Foundations of Software Engineering, ACM, **(2005)**.
[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill and D. R. Engler, "EXE: automatically generating inputs of death", In Proc. ACM CCS, **(2006)**, pp. 322–335.
[10] Yan J. and Zhang J., "An efficient method to generate feasible paths for basis path testing[J]", Information Processing Letters, vol. 107, no. 3, **(2008)**, pp. 87-92.
[11] Xie T., Tillmann N. and de Halleux J., "Fitness-guided path exploration in dynamic symbolic execution[C]", //Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on. IEEE, **(2009)**, pp. 359-368.
[12] T. Xie, D. Marinov and D. Notkin, "Rostra: A framework for detecting redundant object-oriented unit tests", In ASE 04: Automated Software Engineering, IEEE, **(2004)**, pp. 196–205.
[13] http://help.sap.com/abapdocu_731/en/abennesting_depth_guidl.htm
[14] T. J. McCabe, "A Complexity Measure[J]. Software Engineering", IEEE Transactions on, vol. 2, no. 4, **(1976)**, pp. 308-320.

