ICIC Express Letters ICIC International ©2016 ISSN 1881-803X
Volume **10**, Number **2**, February **2016** pp. **323–330**

A NOVEL APPROACH TO MINE SOFTWARE SIMILAR EXECUTION PATHS BASED ON NODE RANK

Haitao He, Yang Liu*, Jun Dong, Jiadong Ren
Hongfei Wu and Yanling Li

College of Information Science and Engineering
Yanshan University
No. 438, Hebei Avenue, Qinhuangdao 066004, P. R. China
haitao@ysu.edu.cn; *Corresponding author: liuyang_ysu@163.com

Received July 2015; accepted September 2015

Abstract. *Analyzing dynamic execution path is very useful to find similar execution paths in software to reduce software testing cases. In this paper, a novel approach is proposed to mine similar execution path to improve test efficiency. First all function calls in software are mapped to nodes sequence by tracking software dynamic execution process. Each node is assigned an initial integer to represent node order for function call. Then the Software Node Rank Model (SNRM) is constructed. Next we use GCC and GNU to recompile open source package to add debugging information to software execution path. Due to loop, sequences maybe appear multiple times, and an algorithm is designed to eliminate repetitive sequence pattern. By Exclusive Nor Operation (XNOR), we can get the similarity among execution paths from binary sequence number corresponding with nodes. When the similarity exceeds given threshold, we regard them as the similar test path. Experimental results show efficiency of the method.*
Keywords: Execution path, Similarity, Exclusive Nor Operation (XNOR), Sequence mining

1. **Introduction.** Software testing is common technique for validating quality of software, how to reduce the number of cases in software testing process has become a pressing matter of the moment [1]. Analyzing similar execution path can greatly reduce test search space.

There are defects of the traditional test method. Equivalence testing notices data dependence relation and function itself. By these technologies, we give more consideration to the judgment skills. Decision table technique considers not only data, but also logical dependencies [2]. However, it cannot express the action of repetitive execution. Boundary testing can find a number of errors through control input or output boundary value, but it needs a long test experience [3]. All functional test methods have limitations: there are loopholes and redundant testing untested. And cost of structural testing is very large.

Program execution path is closely associated with the test cases. By designed cases, Shan et al. [4] put forward dynamic method to test target program and transform judgment statements to Boolean assignment statements. However, it cannot cover all execution paths. Chernak [5] stressed the evaluation of effectiveness of test case, but he did not consider other monitoring information, such as test coverage. Zhang and Gong [6] used search space method to reduce test data for path analysis. In process of path selection, artificial analysis method is used. On the contrary, static analysis can obtain all solutions of test cases. Symbolic execution is the most commonly used method of static analysis. It introduced symbolic into program input to build constraint system to detect the symbolic execution. However, symbolic execution cannot be used to analyze the loop variable and array table structure accurately [7]. King [8] first introduced method of symbolic execution to debugging process, and the experimental results show that method

for debugging sequential execution of program can achieve better effect. Combined with symbolic execution, Young and Taylor [9] proposed a method for detecting program concurrency. Taylor algorithm was used to generate a group program flow graph, and then symbolic execution path was expressed by path expression to detect deadlock in program. However, the above methods are sensitive for path.

From the perspective of data mining, each path of software execution can be considered as a sequence [10]. Sequential pattern mining was first proposed by Agrawal and Srikant [11], and they put forward the Apriori algorithm to find frequent itemsets in association rules. Based on the Apriori algorithm, three kinds of improved algorithm [12] were presented, which are Apriori All, Apriori Some, Dynamic Some. They improved efficiency of Apriori algorithm. However, these algorithms must scan database many times. Thus, their time and space costs are very large. Therefore, Han et al. proposed a FP-Growth algorithm based on frequent pattern tree. In SPADE [13], equivalence classes were put forward to carry on some simple connection to replace multiple scanning databases. Although SPADE can greatly improve time efficiency, a lot of candidate sets would be generated. The similar execution path often appeared in high frequent degree candidate sets [14], so we can filter useless candidate sets with low frequent degree to reduce the input.

In this paper, we propose a novel approach to mine software similar execution paths. First we transform called function nodes in the software dynamic execution process as rank sequence to establish SNRM. Then, we put forward the algorithm to get software execution sequence. Finally, we use Exclusive Nor Operation to compute similarity between patterns to obtain similar execution paths.

The rest of this paper is organized as follows. Section 2 introduces definitions and constructs software node rank model. In Section 3, a method to find similar execution paths is described. The experimental results on open source software are given in Section 4. Finally we conclude this paper.

2. Preliminaries.

2.1. Definitions.

In the software dynamic execution process, software function calls can be seen as sequences.

Definition 2.1. *Execution sequence (ES). Execution sequence is defined as* $S < S_i, S_{i+1}, \ldots, S_j >$, S_i *is an event of* S *and it is expressed as a triple (Fname, Faddr, Flg), where Fname represents the function call name, Faddr represents the memory address of the calling function, if a function starts executing, Flg is E, while the function exits, Flg is X.*

Sequence pattern (SP) is a fragment of sequence S defined as $SP < S_n, \ldots, S_m >$, *where Sn.Flg = E, Sn.Addr = Sm.Addr, Sn.Fname = Sm.Fname, Sm.Flg = X. That is to say, Sn and Sm are the same node. Therefore, SP begins to call Sn and ends to exit Sn.*

Definition 2.2. *Mapping function F (V → I). F (V → I) is a mapping function. V represents node in software execution sequence. The I is an increasing integer from 0 to N. N is maximum value for node contained by the software system.*

Definition 2.3. *Characteristic sequence value (CSV). CSV is a sequence composed of 1 or 0 which represents functions' status in executing process.*

The size of CSV is the number of function nodes contained in the software. In S each bit represents a node, if corresponding node exists in the execution sequence, the bit of CSV is 1; otherwise it is 0.

Definition 2.4. *Sequence similarity.*

Sequence similarity is defined as follows.

$$Sim(Spa, Spb) = \frac{|Spa \odot Spb|}{|S|} \tag{1}$$

Exclusive Nor Operation (XNOR) is denoted by $\odot$. Spa and Spb represent different sequences, $|Spa \odot Spb|$ says the number of 1 in the results after Exclusive Nor Operation for Spa and Spb, and S means the size of CSV. The value of Sim(Spa,Spb) should be less than 1. The higher Sim value is, the more similar sequences are.

2.2. **Software node rank model constructing.** First we get the value of N which is the number of nodes obtained by traversing overall the software. At the same time, every node will be given an initial value as the node label. The Label I is an incremental integer, thus the mapping function F ($V \rightarrow I$) is also generated, and the algorithm is described as follows.

In Algorithm 1, line 1 to line 4 traverses nodes to establish the mapping F ($V \rightarrow I$). Line 5 shows initialization of the CSV. Through the pretreatment function node of the software system, we get the mapping function F and CSV on line 6.

Algorithm 1 SNRM Constructing

Input: Nodes generated by software execution process
Output: The mapping function F and characteristic sequence value (CSV)

```
1: Given a global variable I
2: The initialization of I is 0
3:   for each node set I
         if (I is 0)
           Set I = 1;
         else I++;
4:   end for
5: initialize CSV, the size of CSV is I;
6: return F, CSV
```

3. Approach to Mine Similar Execution Paths.

3.1. **Sequence generating and repetitive pattern eliminating.** The initial data is generated in software execution process. It only contains function's address in memory with entrance-exit flag. At this phase, we add corresponding function name to function address in software execution process, and sequence unit as triple (Fname, Faddr, Flg) will be produced.

Due to the complexity of software execution, software execution sequence is very complex. Because of cycle in the function call process, a function would appear 10 times circulation as 10000 calls. Therefore, if repetitive patterns are eliminated, time and space efficiency in analysis process will be improved.

Algorithm 2 has two procedures. Line 1 to line 3 shows how to generate sequence of execution. The return of the sequence of S contained three tuples (Fname, Faddr, Flg). Line 4 to line 10 is to search out all sequence fragment of software execution sequence. Then, Line 11 to line 13 will make clear the pseudo code for eliminating repetitive sequence fragment.

Algorithm will add segments in stack which appears first time and other segments of repetitive sequence will be deleted.

3.2. **Reduced sequential patterns producing.** In Algorithm 2, we get software execution sequence after eliminating repetition.

We will convert function addresses to called function nodes; thus, software pattern set of function calls will be produced. In Algorithm 3, line 1 to line 9 judges whether sequence between two events in RSP is a pattern or not, and then it will be stored in PS. We can get pattern set PS on line 10.

Algorithm 2 Sequence generating and eliminating repetitive patterns

```
Input: execution sequence (ES), event Si with Flg and Addr
Output: Reduced Sequential Pattern RSP
 1: Sort different and non-redundant events from ES and store them in Addr Stack
 2:   for each event Si in ES do
        if (AddrStack exists Si.Addr)
          Store (Si.Flg, Si.Addr, addr to function (Si)) in S;
 3:       for each Si in S do
            if (Si.Flg = = 'E')
 4:           for each Sj in S do
                if (Sj.Flg = = 'X' and Si.Fname = = Sj.Fname)
                  Replace (Pi, P (Si, Sj)) in S';
                   break;
 5:             end if
 6:           end for
 7:         end if
 8:       end for
 9:     end if
10:   end for
11:   for each Pi in S' do
        if(exists i consecutive pattern Pi in S')
          retain P1;
        then delete other P in S', until S' is stable;
          Replace (event sequence, each P in S');
        then RSP = S';
12:     end if
13:   end for
14: return RSP
```

Algorithm 3 Software sequential pattern mining

```
Input: Reduced Sequential Pattern RSP
Output: Pattern Set PS
 1: for each event RSP do
 2:   if (event.Flg = = 'E')
 3:     for each other event in RSP do
 4:       if (event'.Flg = = 'X') and (P is a pattern and P/PS)
 5:         PS.push (P);
            event = next event;
 6:       end if
 7:     end for
 8:   end if
 9: end for
10: return PS
```

3.3. **Similar execution path finding.** After sequential pattern mining, we get patterns of software execution sequence. Then similarity is calculated based on the software pattern sets, and the calculation consists of three steps as below.

Step1. Find the corresponding value of the execution sequence mapping software node rank model.

Step2. Get the characteristic sequence values (CSV) of every execution sequence, and the size of CSV.

Step3. Calculate the similarity using the XNOR respectively.

For instance, assume that P1 = A, B, C, G, H, P2 = A, B, D, E, F, G, H, P3 = A, B, C, D, E, F, G, where A, B, C, D, E, F, G, H are patterns.

First, we find out P1, P2 and P3 corresponding value from SNRM.

Table 1. Software node rank model for example

node	A,	B,	C,	D,	E,	F,	G,	H
value	1,	2,	3,	4,	5,	6,	7,	8

In software node rank model, we can get the rank of each node: (A $\rightarrow$ 1), (B $\rightarrow$ 2), (C $\rightarrow$ 3), (D $\rightarrow$ 4), (E $\rightarrow$ 5), (F $\rightarrow$ 6), (G $\rightarrow$ 7), (H $\rightarrow$ 8).

Second, we get Sp1 = (11100011), Sp2 = (11011111), Sp3 = (0111111), and the size of CSV is $|S| = 8$.

Third, the similarity between Spa and Spb is Sim(Sp1, Sp2) = ($|11100011 \odot 11011111|$) /8 = 4/8, Sim(Sp2, Sp3) = ($|11011111 \odot 11111110|$)/8 = 6/8, Sim(Sp1, Sp3) = ($|11100011 \odot 11111110|$) /8 = 4/8. We assume the threshold is 0.75. Therefore, Sp2 and Sp3 will be merged.

4. **Experiment and Analysis.** In this section, we test algorithms on two open source software coded in C or C++, including C program analysis tool Cflow for the relationship of function calls, and a free software GNU file compression program Gzip. On Linux operating system, GCC is used to recompile them by adding debug information, to gather the information of program traces automatically and obtain the software execution sequence. Here, we select five test cases to perform Cflow and Gzip respectively. In each trial, we transform software execution process as pattern sequences and eliminate repetitive patterns. We test cases five times on each software. Therefore, five pattern collections can be obtained. Tables 2 and 3 respectively show Cflow and Gzip's software node rank model.

In Tables 2 and 3 we can see that every node in Cflow and Gzip corresponds to an integer, and the integer is node order in sequence table. First nodes in execution sequence are obtained, and then corresponding binary sequences are generated.

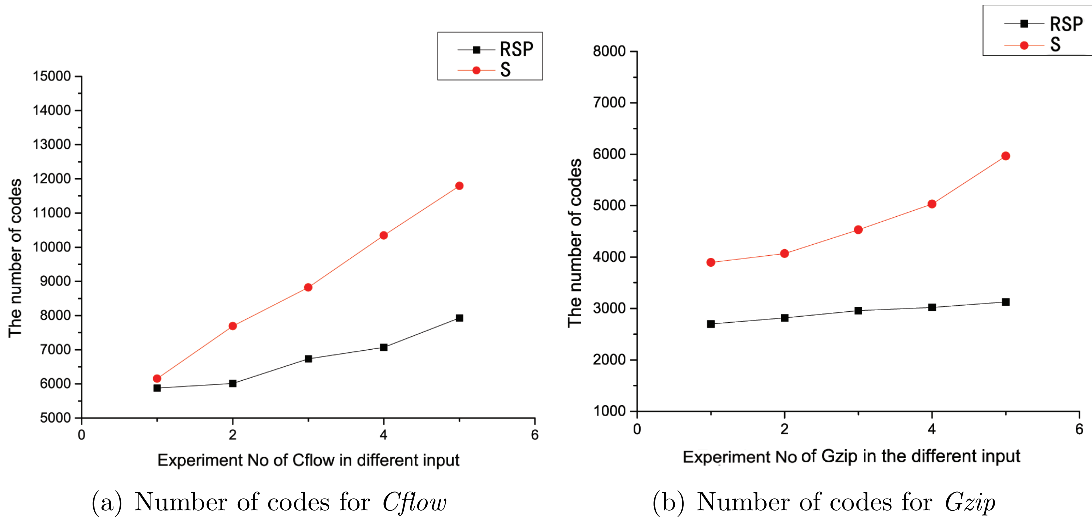
Table 2. The SNRM of Cflow

Node	Value
main	1
cregister_output	2
sourcerc	3
parse_rc	4
parse_opt	5
add_name	6
append_to_list	7
alloc_cons	8
alloc_cons_from	9
...	...

Table 3. The SNRM of Gzip

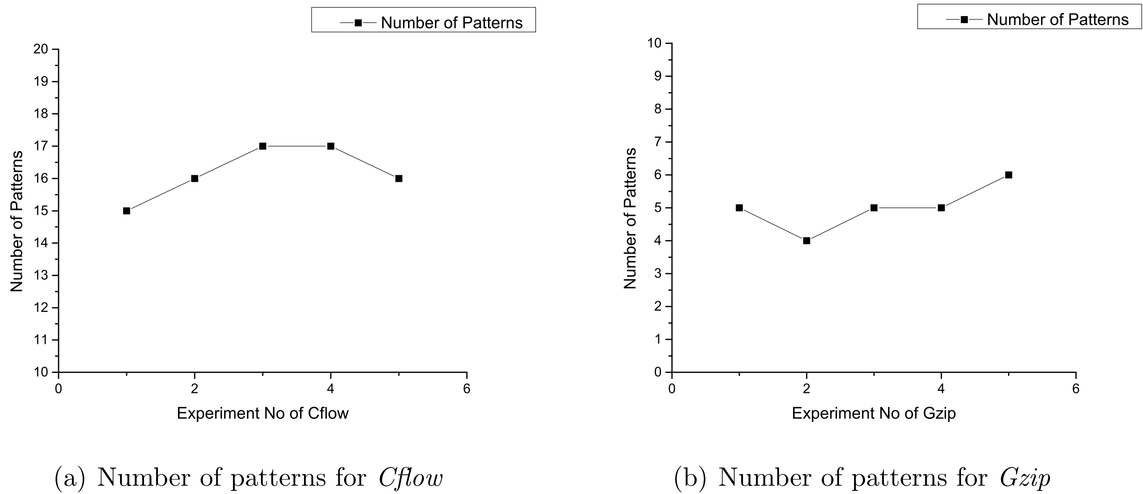
Node	Value
main	1
treat_file	2
open_input_file	3
build_tree	4
zip	5
ct_init	6
gen_codes	7
file_read	8
read_buffer	9
...	...

Figure 1 describes trend comparison between lengths of original software execution sequence and lengths of Reduced Sequential pattern (RSP) for Cflow and Gzip in five experiments respectively. In Figure 1(a), with the increasing of software execution sequence length, lengths of RSP also show growing tendency, but growth rate is relatively small. By the same token, in Figure 1(b), with the increasing of software execution sequence length, amplitude of variation length of RSP is very small.



(a) Number of codes for *Cflow* (b) Number of codes for *Gzip*

FIGURE 1. The number of codes of Cflow and Gzip



(a) Number of patterns for *Cflow* (b) Number of patterns for *Gzip*

FIGURE 2. The number of patterns of Cflow and Gzip

TABLE 4. The number of patterns in 5 experiments for Cflow and Gzip

	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5
Cflow	15	16	17	17	16
Gzip	5	4	5	5	6

Figure 2 describes trend comparison for lengths of pattern sets (PS) for Cflow and Gzip in five experiments respectively. In Figure 2, we see that with the increase of software execution sequence length, changes of patterns are small for both Cflow and Gzip.

Table 4 shows the number of patterns in different pattern sets obtained by 5 tests on Cflow and Gzip. From the results we can see that changes of the number of sequence patterns are very small. Table 5 and Table 6 show the 5 test results of S(N) between Cflow and Gzip, where S denotes the similarity and N indicates the number of identical sequence in patterns.

Table 5 and Table 6 show similarities between sequence patterns obtained in 5 experiment results of Cflow and Gzip. Table 5 shows that range of similarity for Cflow is from 0.65 to 1. Table 6 shows range similarity for Gzip. In experiments, we set threshold as

Table 5. The similarity of patterns for Cflow

	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5
Exp 1	1.00(15)				
Exp 1	0.71(12)	1.00(16)			
Exp 1	0.76(13)	0.76(13)	1.00(17)		
Exp 1	0.71(12)	0.76(13)	0.82(14)	1.00(17)	
Exp 1	0.65(11)	0.71(12)	0.76(13)	0.71(12)	1.00(16)

Table 6. The similarity of patterns for Gzip

	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5
Exp 1	1.00(5)				
Exp 1	0.60(3)	1.00(4)			
Exp 1	0.80(4)	0.60(3)	1.00(5)		
Exp 1	0.60(3)	0.60(3)	0.60(3)	1.00(5)	
Exp 1	0.80(4)	0.80(4)	0.80(4)	0.60(3)	1.00(6)

0.8, about 20% tests can be reduced for Cflow, and about 40% tests can be reduced for Gzip.

5. **Conclusions.** In this paper, a novel approach is proposed to mine similar execution path. First, it transforms called function nodes in software dynamic execution process as rank sequences to construct SNRM. By F ($V \to I$), each node is assigned an initial integer to represent node order for function call. Then, in our eliminating repetitive algorithm, we remove those repeated calls to optimize the structure of sequential patterns and reduce energy consumption of experimentation. Calculating similarity between sequences in matching process by XNOR, we find that execution process for same software is basically same or similar. Therefore, according to proposed method, we can divide software dynamic execution process into a number of stable patterns. And those stable patterns often represent similar execution paths. Through analysis of these stable software patterns, we can merge those similar execution paths who meet threshold condition. Experiment results show that it can facilitate us to reduce test case in software testing. In future work, we aim to find the correlation relationship between nodes based on matching of software sequence pattern. So that, combining the similarity between nodes and sequence structures, we can more accurately find out similar execution path in software dynamic execution process.

Acknowledgment. This work is supported by the National Natural Science Foundation of China under Grant No. 61170190, No. 61472341 and the Natural Science Foundation of Hebei Province China under Grant No. F2013203324, No. F2014203152 and No. F2015203326. Authors also gratefully acknowledge the helpful comments and suggestions of reviewers, which have improved the presentation.

REFERENCES

[1] K. Zhou, J. Feng, W. Lan et al., Cluster analysis of software testing paths based on complex networks, *Computer Engineering and Applications*, vol.46, no.31, pp.72-76, 2010.

[2] J. Ye, A method to generate the most similar path set based on given failure path, *IEEE International Conference on Intelligent Computing & Intelligent Systems*, pp.635-638, 2010.

[3] D. K. Sharma, R. K. Sharma, B. K. Kaushik and P. Kumar, Boundary scan based testing algorithm to detect interconnect faults in printed circuit boards, *Circuit World*, vol.37, no.3, 1974.

[4] J. Shan, J. Wang and Z. C. Qi, Survey on path-wise automatic generation of test data, *Acta Electronica Sinica*, vol.32, no.1, pp.109-113, 2004.

[5] Y. Chernak, Validating and improving test-case effectiveness, *Software IEEE*, vol.18, no.1, pp.81-86, 2001.

[6] Y. Zhang and D. W. Gong, Evolutionary generation of test data for path coverage based on automatic reduction of search space, *Acta Electronica Sinica*, vol.40, no.5, pp.1011-1016, 2012.

[7] A. Saswat, C. S. Păsăreanu and W. Visser, Symbolic execution with abstraction, *International Journal on Software Tools for Technology Transfer*, vol.11, no.1, pp.53-67, 2009.

[8] J. C. King, Symbolic execution and program testing, *Communications of the ACM*, vol.19, no.7, pp.385-394, 1976.

[9] M. Young and R. N. Taylor, Combining static concurrency analysis with symbolic execution, *IEEE Trans. Software Engineering*, vol.14, no.19, pp.1499-1511, 1988.

[10] S. A. Ebad and M. A. Ahmed, Functionality-based software packaging using sequence diagrams, *Software Quality Journal*, pp.1-29, 2014.

[11] R. Agrawal and R. Srikant, Fast algorithms for mining association rules, *Proc.of 1995 Int. Conf. Data Engineering ICDE*, Taipei, Taiwan, pp.3-14, 1995.

[12] R. Agrawal and R. Srikant, Mining sequential patterns: Generalizations and performance improvements, *Proc. of the 5th Int. Conf. Extending Database Technology*, 1996.

[13] M. J. Zaki, Fast mining of sequential patterns in very large databases, *Technical Report*, vol.668, 1997.

[14] J. D. Ren, Y. F. Tian and H. T. He, Bitmap-based algorithm of mining approximate sequential pattern in data stream, *Journal of AISS*, 2011.