Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 14, no. 6, June 2018, pp. 1181-1189
DOI: 10.23940/ijpe.18.06.p9.11811189

Software Reliability Test Case Generation using Temporal Motifs Recovery and Configuration

Xuetao Tian*, Feng Liu, and Honghui Li

[a]*School of Computer and Information Technology, Beijing Jiaotong University, Beijing, 100044, China*
[b]*Engineering Research Center of Network Management Technology for High Speed Railway of MOE, Beijing, 100044, China*

Abstract

In software updating process, software reliability test plays an important role. Test cases are the key to software test. To match real usage habit, log analysis becomes a hot way to generate reliability test cases. However, using log analysis can't cover new operations arising from software updating. In this paper, a reliability test cases generation method for updating software using temporal motifs recovery and configuration is presented in this paper. We tentatively introduce temporal network idea to abstract software usage log. Test cases adapting to software updating are generated using temporal motifs recovery and configuration. As a case study, the method is applied to an online application. The coverage frequency comparison experiment is designed. The proposed method can obtain similar results as log and Markov model. Thereby, the usability of the method is validated.

Keywords: software updating; reliability test cases; log analysis; temporal motifs recovery; motifs configuration

(Submitted on February 27, 2018; Revised on April 1, 2018; Accepted on May 21, 2018)

1. Introduction

Software reliability test is a software fault-facing test method, where its core idea is to test the software according to the users' habits and quantify software reliability with some indexes, such as failure rate and mean time between failures, in the expected running environment [13]. The key to software test is to generate appropriate test cases. Obviously, the more similar to users' habits test cases are, the more precise reliability evaluation is. In previous studies, software reliability test generally depends on a test model describing the software usage situation called model-driven test method. There are two main kinds of model-driven test methods: usage model-based and profile-based test method. Both consist of specific structure and attached parameters derived from statistical regularities in actual usage situation. Reliability test cases can be automatically generated using these models [6].

For a long-time, running software like web application accumulated a mass of log data. Log analysis has been used for modeling in many other areas. The software log contains all users' operations to the software and implies the usage habits. Thus, log analysis becomes an innovative way to conduct reliability test. Software reliability test based on log can't only implement automatic test, but also have a more precise description of usage habit than empiricism.

The idea of temporal motifs comes from structure analysis of temporal network involving topology and time attribute [10]. Temporal motifs simultaneously consider subgraph isomorphism and time series restriction [12]. Obviously, software log can be abstracted into temporal network, which is the most common Markov usage model for reliability software test represented as a graph structure. Test cases generated from one model are actually the operation sequences in time dimension, which contain the temporal motifs in log. However, the limitation of log analysis is that it can't generate test cases that cover new operations arising from software updating.

* Corresponding author.
E-mail address: xttian@bjtu.edu.cn

This paper presents an exploratory method for software reliability test cases generation using temporal motifs recovery based on log analysis. This method takes log as a temporal network model and applies motifs recovery idea to test cases generation. Reliability test cases will compose of these frequent temporal motifs. This paper also proposes motifs configuration to match software updating situation. We apply this method to an online application. The coverage frequency of the important operations serves as the index. The proposed method is compared with log and Markov model, and the usability of this method is validated.

The rest of this paper is organized as follows: Section 2 states the related works on software reliability test, involving software test using log analysis, updating software test and temporal motifs research. The method presented by this paper is elaborated in section 3. Section 4 gives you a case study and validates that the method in this paper can generate test cases with similar operations coverage frequencies as log and Markov model in software updating process. Conclusions and perspectives are put forward in section 5.

2. Related works

Software reliability means the probability that the software doesn't cause system failure under specified conditions and within a specified time. The probability is a function of the system input and usage, as well as a function of the defects existing in the software. The system input will determine whether existing defects will be encountered. The American Institute of Standardization approved this standard as American national standards [1] and China national standard GB/T-11457 [4] adopted it in 1989. Previous researchers have done sufficient work on software reliability test. The core step of software reliability test is test cases generation. Test cases should be in accordance with software usage habit. There are two kinds of statistical reliability test methods, operation profile-based method proposed by Musa [11] and Markov usage model-based method raised by Mills and Whittaker [9]. Both simulate the software usage habit in the test process. Markov model can be represented as a directed graph with probability, and reflects software usage habit with statistical thought [3]. However, software is dynamically in use.

Log analysis is an innovative way to generate reliability test cases since log reflects real users' habit. Previous researchers [2,13,14] did studies on software test based on log analysis. Mette Arleth uses log analysis for testing cartographic web-based application. X. Tian constructs software reliability test model through log analysis and generates test cases using this model. Some researchers in other filed did some works to recover pattern from original records [7,8]. However, it can't overcome the limitation of log analysis that is not testing those new operations arising from software updating.

With software updating, the usage frequencies of original operations remain generally stable. Tab. 1 testifies to it, which calculates top-5 most frequently used operations in every version of selected application. Thus, updating software reliability test cases can be generated by analyzing log.

Table 1. The top-5 most frequently used operations in every version of selected application

Top	Ver. 0	Ver. 1	Ver. 2	Ver. 3	Ver. 4	Ver. 5	Ver. 6	Ver. 7	Ver. 8	Ver. 9	Ver. 10
1	a	a	a	a	a	a	a	a	a	a	a
2	b	b	b	b	b	b	c	c	c	c	c
3	c	c	c	c	c	c	b	b	b	b	b
4	d	e	d	e	d	d	d	d	d	d	d
5	e	d	e	d	e	f	e	e	e	e	g

1. The data is from the dataset in section 4.1.
2. There is continuous eleven versions of this application in this dataset.
3. In the table, *a*, *b*, *c*, *d*, *e*, *f* and *g* represent separately operation *4fae340ca4746b3b75b683beb97518be*, *8edda9a7b0ed9cc689ec845fdf36f9ea*, *253d068b16b61e8d4bea5cf9bada1a8e*, *3242b6aae2fed3b880f7fa93813a5d35*, *28ad5473fa7f24dc53fb1e0061866002*, *3dd98db793d7cc80f1a8da3e00663e31*, *480970d8c149055ec554c7c9ae0a2d5d* occurring in this dataset.

This paper abstracts log as a model based on the temporal networks thought. Petter Holme [5] describes temporal networks as follows. Temporal networks are extensions of static networks. They are based on a mathematical structure that links entities pairwise, but the structure captures information about a pair of nodes interaction and the time of the interaction in temporal networks. In other words, temporal networks cover the time dimension on the basis of graph structure. Temporal motifs are defined as induced subgraphs on sequences of temporal edges. Mining motifs is an effective way to understand temporal networks. Considering that test cases is the sequence of users' operation in the time dimension, the temporal motifs and the test cases have the same data structure expression. On the basis of previous works, this paper tentatively applies temporal motifs recovery and configuration to software reliability test cases generation. It can conduct reliability test for

updating software using log analysis. At the same time, using motifs configuration overcomes the disadvantage of log analysis that new operations can't be covered.

3. Updating software reliability test cases generation method

In essence, an operation in software log indicates software state transition. Those operations arising from software updating will certainly establish a connection to one or more previous operations, which enlightens the way of using artificial configuration to match software updating. This section elaborates reliability test cases generation method in the process of software updating. Before the specific description, the overview of the method and motifs recovery algorithm will be stated.

3.1. Overview of the method

As shown in Fig. 1, the proposed method in this paper has four stages: preprocessing, motifs recovery, motifs configuration and test cases generation.

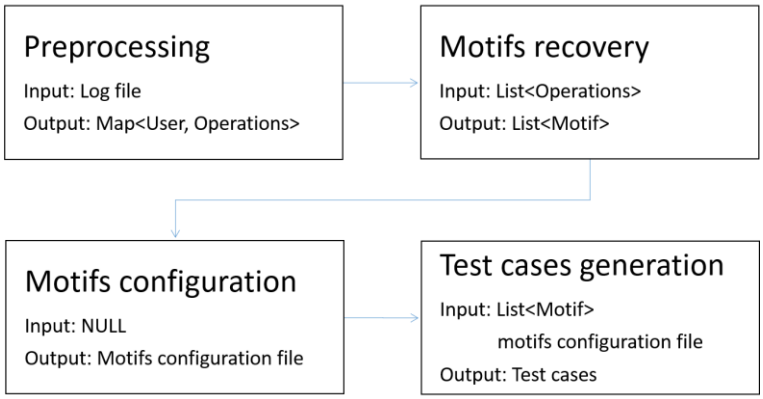


Figure 1. Overview of the proposed method

Preprocessing. In the preprocessing, the input is original log file. Those unrelated and wrong records will be removed. Then, some necessary information are encrypted to generate the dataset in section 4.1. Based on the dataset, the log is split up into 11 parts in accordance with the updating timestamp. When test cases generation is based on version *i*, the corresponding part is processed into a number of operation sequences for each user, which are stored by a Map<User, Operations> structure and serve as the output.

Motifs recovery. For each operation sequence, all motifs will be dug out. To reduce the amount of calculation, the maximum length of a motif is set 10. The motifs recovery algorithm will be introduced in section 3.2. The motifs with not very low frequency are chosen as the output to generate test cases.

Motifs configuration. This is an artificial work. In the process of software updating, some new operations will occur, where each operation will establish a connection to one or more previous operations. Motifs configuration file is built artificially to keep an account of these connections from version *i* to version *(i+1)*. Operation cancellation will also occur, but it isn't presented in this paper since it has a similar principle.

Test cases generation. On the basis of previous work, the motifs in stage 2 can be modified for matching software updating using motifs configuration. Thereafter, these new motifs are combined into test cases based on some certain rules.

3.2. Temporal Motifs Recovery

The structure of temporal networks is defined as a set of nodes and a collection of directed temporal edges, where each edge has a timestamp. According to Ashwin Paranjape's study [12], a small temporal network with four nodes and eight temporal edges is illustrated in Fig. 2, where edges is between five ordered pairs of nodes.

The definition of temporal motifs extends from network motifs, which are small induced subgraphs occurring in a bigger network structure. A δ-temporal motif is a given network motif M where all the edges occur inside the time period of

δ time units. Fig. 3 shows a motif structure with three nodes and three edges where the edge label denotes the occurring order. All motifs matching this structure and δ = 10s are also shown.

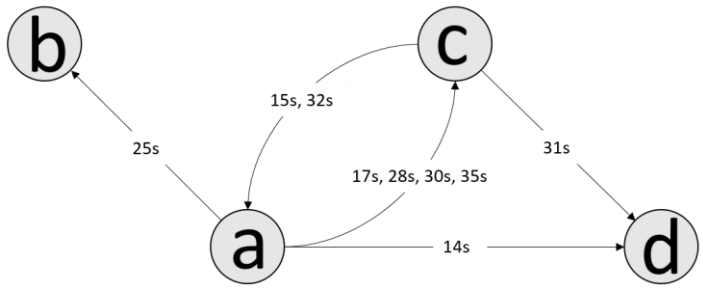


Figure 2. A small example of temporal networks

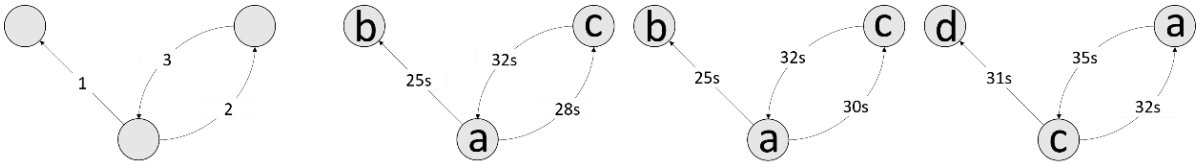


Figure 3. A motif structure and matched motifs in Fig. 2

For a software, log which records users' operations can be abstracted as a temporal network, where a node and an edge respectively represent specific operation and transfer between operations. Nevertheless, software log is dissimilar from the general situation. An edge in a temporal network represents the relation between two nodes with the occurring timestamp, but log file records a mass of continuous processes. For precise description, some limitations are given to the temporal network definition. The following proposition is given.

PROPOSITION 1. *Complete software operations in one usage can be abstracted as a temporal network with* N *nodes, where the following conditions are satisfied:*
(1) *The static isomorphism graph of this network is a connected graph.*
(2) *For* (N - 2) *nodes, each of them has the same in-degree and out-degree.*
(3) *For another node, in-degree minus out-degree is one.*
(4) *For the last node, out-degree minus in-degree is one.*

Based on Proposition 1, a LIST structure is used to represent a software operations sequence. Assuming that there is an operation sequence without time restriction, we denote it as *panel*. Alg. 1 elaborates how to extract motifs from a *panel*. On this basis, Alg. 2 describes how to dig out all motifs from an operations sequence with LIST representation. Software reliability test requires test cases matching users' habit. Motifs with length 1 and 2 can't do it, which just contain some information about software operations. Thus, motifs with length least 3 are recovered in the algorithm.

Algorithm 1 Extract motifs from panel

```
1: Required: an operations sequence without time restriction: panel
2: assign motifs ← {};
3: assign n ← panel.size;
4: assign motifs_matrix ← a matrix with n * n;
5: for all i in 1 to n – 1 do
6:     for all j in 0 to n – i + 1 do
7:         if j == 0 then
8:             assign motifs_matrix[j][j+i] ← motif with length 2 composed of operation j and j + i;
9:         else
10:            assign motifs_matrix[j][j+i] ← {};
11:            for all k in 0 to j – 1 do
12:                assign motifs_matrix[j][j+i] ← motifs_matrix[j][j+i] + combine each operation of motifs_matrix[k][k+i]
```

```
                with operation j;
13:           end for
14:           assign motifs ← motifs + motifs_matrix[j][j+i];
15:       end if
16:   end for
17: end for
18: Output: All motifs in panel: motifs
```

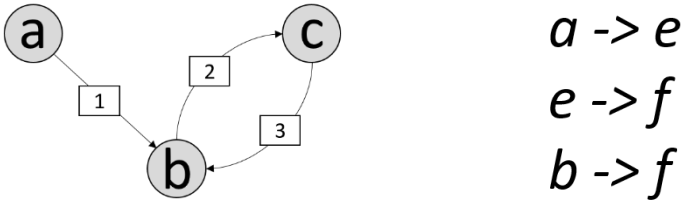
Algorithm 2 Motifs recovery in an operations sequence

```
 1: Required: an operations sequence with LIST representation: O
              time units restriction: δ
              maximum motifs length: max_length
 2: assign motifs ← {};
 3: assign start_operation ← the first operation of O;
 4: assign panel ← {start_operation};
 5: for all operation in O do
 6:   if operation.timestamp – start.timestamp > δ or panel.size >= max_length then
 7:     if panel.size >= 3 then
 8:       assign motifs ← motifs + ExtractMotifsFromPanel(panel);
 9:     end if
10:     assign start_operation ← the next operation of start_operation in O;
11:     assign panel ← {start_operation};
12:   else
13:     assign panel ← panel + operation;
14:   end if
15: end for
16: Output: All motifs in O: motifs
```

3.3. Test cases for updating software

Before introducing the test cases generation method, the fusion method for motifs and motifs configuration file will be explained. As shown in Fig. 4, there is an artificial example of motif and relevant motifs configuration, where *a*, *b* and *c* are original operations and *e* and *f* are operations arising from software updating. It is $a \rightarrow e$ that represents *e* can be operated after operation *a*.

The motif in Fig. 4(a) is denoted as a sequence $a \rightarrow b \rightarrow c \rightarrow b$. When using the motifs configuration to update motifs, the configuration is modified, which focuses on the configuration information between new operations, such as $e \rightarrow f$ in Fig. 4(b). After this processing, configuration $\{a \rightarrow e, e \rightarrow f, b \rightarrow f\}$ will be modified to $\{a \rightarrow e, a \rightarrow (e, f), b \rightarrow f\}$, where $a \rightarrow (e, f)$ represents that *e*, *f* can be operated continuously after operation *a*.



(a) Motif (b) Motifs configuration

Figure 4. An instance of motif and relevant configuration

In the result of motifs recovery, a frequency value is attached to each motif. Assume the frequency of $a \rightarrow b \rightarrow c \rightarrow b$ is *N*. Fig. 5 intuitively displays the fusion process. For each configuration item, the origin motif will be fused with a random probability α, β, or γ and retained with 1 - (α + β + γ). For the fusion of $a \rightarrow e$ and $a \rightarrow b \rightarrow c \rightarrow b$, it only needs to replace the back of a, $b \rightarrow c \rightarrow b$, by *e*. However, for the fusion of $b \rightarrow f$ and $a \rightarrow b \rightarrow c \rightarrow b$, it needs to choose which *b*'s back is replaced by *f*. Generally, δ in Fig. 5 equals 0.5 and α, β, γ will be a number far less than 1. But it is not absolute. Finally, the

motif $a \rightarrow b \rightarrow c \rightarrow b$ is modified into 5 motifs in Fig. 5 with frequency values $N * \alpha$, $N * \beta$, $N * \gamma * \delta$, $N * \gamma * (1 - \delta)$ and $N * [1 - (\alpha + \beta + \gamma)]$.

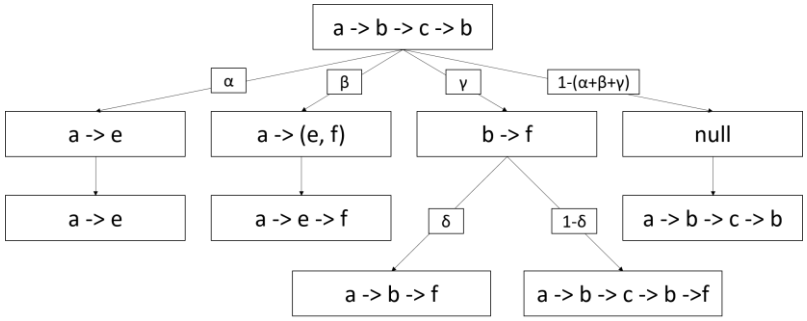


Figure 5. The fusion process for motif and motifs configuration in Fig.4

Based on the above method, new motifs are obtained. Through randomly connecting these motifs, reliability test cases can be generated. In this process, it needs to be guaranteed that the last operation of the former motif is the same as the first one of the latter motif. In the next section, a case study using this method is given.

4. Case study

The proposed method is applied to an online application. We collate the log into a dataset. An instance of motifs recovery and motifs configuration will be provided. The experiment to compare the frequencies of selected operations is designed to validate the usability of this method.

4.1. Dataset

The usage logs of one online smartphone application are used. We pre-processed parts of these records to retain effective information and hide necessary information. Finally, the dataset contains 4,356,066 usage records of the selected software covering continuous 11 versions, where 148 unique operations occur and each record contains TIMESTAMP, USER-ID and OPERATION-ID information. The overview of the dataset is shown in Tab. 2. Meantime, the update timestamps and artificial corresponding motifs configurations are given in the dataset. The dataset can be provided freely for research purposes.

Table 2. The overview of the dataset

Version	Start-Time	End-Time	Record-Num	New-Operation-Num	User-size
0	1480545167	1481052314	173497	--	5843
1	1481052330	1481927153	289940	6	8563
2	1481927160	1483215044	472565	15	9981
3	1483215060	1484160842	324118	2	7222
4	1484160850	1485615550	555109	5	11454
5	1485615561	1485796989	58611	0	1842
6	1485797006	1487095881	442656	5	10298
7	1487095905	1488728911	741866	12	13456
8	1488728933	1489784150	503227	0	9727
9	1489784171	1490369664	276449	2	5301
10	1490369673	1491432442	518028	3	9319

4.2. An instance of motifs configuration

Operation *f996acabde4001f6aa876ced222269bf* (*f99*) arising from version 6 updating is selected as the instance object owing to some unconventional cases. [*4fae340ca4746b3b75b683beb97518be* (*4fa*), *046a69840dd0369db8623caa0048c36e*, *046a69840dd0369db8623caa0048c36e*, *bc61a2a685a188ee5b4f9c9c8d45276f*] is a recovered motif in version 6 with frequency 6306 and relevant configuration is {*f99* → *f99*, *a01db36db37910033ca1e4955f0ab73f* (*a01*) → *f99*, *f99* → *a01*, *4fa* → *f99*}, where *a01* is also a new operation in version 7.

According to the algorithm in section 3.3 to modify configuration information, there will be some problems, because {*f99* → *f99*} or {*a01* → *f99*, *f99* → *a01*} can lead to an infinite loop.

For such a situation, the maximal length of a record in configuration is simply set to 4. The modified configuration is {*4fa → f99*, *4fa →* (*f99*, *f99*), *4fa →* (*f99*, *a01*), *4fa →* (*f99*, *f99*, *f99*), *4fa →* (*f99*, *f99*, *a01*), *4fa →* (*f99*, *a01*, *f99*)}. And then, new motifs, as like as the configuration, are obtained.

4.3. Experimental results

In this section, operations are randomly chosen to do frequency contrast aimed at validating the usability of this method. Firstly, the frequencies of these operations in original log are counted, as shown in Tab. 3.

Table 3. The frequencies of selected operations in original log

Op. (Unit)	Ver .0	Ver. 1	Ver. 2	Ver. 3	Ver. 4	Ver. 5	Ver. 6	Ver. 7	Ver. 8	Ver. 9	Ver. 10
a	0.0889	0.0903	0.0915	0.0908	0.0926	0.1004	0.0893	0.0775	0.0711	0.0750	0.0660
b	0.0006	0.0006	0.0009	0.0006	0.0007	0.0007	0.0010	0.0007	0.0006	0.0002	0.0005
c	0.0320	0.0306	0.0355	0.0263	0.0304	0.0280	0.0274	0.0338	0.0343	0.0351	0.0400
d (10^-2)	--	0.0006	0.0012	0.0008	0.0020	0	0.0016	0.0019	0.0014	0.0014	0.0002
e	--	--	0.0052	0.0118	0.0195	0.0223	0.0207	0.0224	0.0272	0.0294	0.0338
f	--	--	0.0003	0.0002	0.0004	0.0002	0.0002	0.0005	0.0002	0.0002	0.0002
h	--	--	0.0048	0.0109	0.0211	0.0271	0.0259	0.0317	0.0334	0.0337	0.0394
i	--	--	--	--	0.0007	0.0006	0.0006	0.0009	0.0008	0.0007	0.0005
j	--	--	--	--	0.0008	0.0009	0.0009	0.0015	0.0014	0.0015	0.0009
k (10^-2)	--	--	--	--	0.0007	0.324	0.0289	0.0245	0.0185	0.0275	0,0178
l (10^-2)	--	--	--	--	--	--	0.0009	0.0040	0.0167	0.0221	0.0149
m (10^-2)	--	--	--	--	--	--	0.0370	0.0359	0.0258	0.0130	0.0249
n (10^-2)	--	--	--	--	--	--	0.0061	0.0230	0.0350	0.0318	0.0344
o (10^-2)	--	--	--	--	--	--	--	0.0096	0.0044	0.0069	0.0041
p (10^-2)	--	--	--	--	--	--	--	0.0360	0.0240	0.0043	0.0091
q (10^-2)	--	--	--	--	--	--	--	0.0005	0.0016	--	--
r	--	--	--	--	--	--	--	--	--	0.0036	0.0165
s	--	--	--	--	--	--	--	--	--	0.0001	0.0001
t	--	--	--	--	--	--	--	--	--	--	0.0060

1. In the table, *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k*, *l*, *m*, *n*, *o*, *p*, *q*, *r*, *s* and *t* represent separately operation *8edda9a7b0e-d9cc689ec845fdf36f9ea*, *baf456cee335d1082a2a8a77a85bf656*, *146423babce0f93bf9e7c367bd3e5ad2*, *a72c0cc7e3ff541ac80d3a4617c9865b*, *7bcb7218c22530fa1b0039f23dbf6872*, *ba11239323ae248e84d58b4bc54646b0*, *6173eb54687f254b2f8f9331340efeaf*, *a18ae1a9d04d9691036d18168ea53a69*, *7453bcdede7753a876bc57b151f83e65*, *1a48e321b732cf9b6e0b0c0428431435*, *e96448cde68b2ae5dc38ec92eaab4270*, *9c23f9ed4e3f91a32c1ee8c448afaa11*, *2d0a969448b84fcb4b69376c199e2b0d*, *f996acabde4001f6aa876ced222269bf*, *0bf2988535271f683eae9cc88ad5beff*, *83765e54d29780d0cf17a885b954aa60*, *686007982a0015a8968032e303be9fb0*, *411162b943adcbc8809a2eba1f20e70f*, *35ded80466dee71ea69c453ccb9d31db* occurring in this dataset.

Then, test cases are generated using the proposed method for each software updating. The scale of each generation is equal to the record number of next version and the maximal length of a test case is set 20. The frequencies of the above operations in these test cases is counted again, as shown in Tab. 4.

Table 4. The frequencies of selected operations in test cases

Op. (Unit)	0 to 1	1 to 2	2 to 3	3 to 4	4 to 5	5 to 6	6 to 7	7 to 8	8 to 9	9 to 10
a	0.0903	0.0934	0.0928	0.0910	0.0947	0.1102	0.0889	0.0767	0.0749	0.0680
b	0.0006	0.0008	0.0010	0.0007	0.0007	0.0008	0.0010	0.0007	0.0005	0.0004
c	0.0315	**0.0170**	0.0356	0.0286	0.0340	0.0267	0.0315	0.0342	0.0346	0.0381
d (10^-2)	0.0010	0.0010	0.0011	0.0010	**0.0010**	**0**	0.0020	0.0030	0.0020	**0.0020**
e	--	0.0087	0.0056	0.0167	0.0187	0.0202	0.0211	0.0285	0.0264	0.0297
f	--	**0.0054**	0.0003	0.0002	0.0004	0.0002	**0.0002**	**0.0005**	0.0002	0.0002
h	--	0.0045	0.0052	0.0118	0.0209	0.0258	0.0254	0.0299	0.0361	0.0392
i	--	--	--	**0.0100**	0.0007	0.0006	0.0006	0.0009	0.0008	0.0007
j	--	--	--	0.0009	0.0009	0.0009	0.0010	0.0016	0.0016	0.0015
k (10^-2)	--	--	--	0.0010	**0.0008**	0.0317	.0276	0.0231	0.0186	0.0244
l (10^-2)	--	--	--	--	--	0.0010	0.0100	0.0010	0.0201	.0200
m (10^-2)	--	--	--	--	--	0.0420	0.0384	0.0321	0.0232	0.0360
n (10^-2)	--	--	--	--	--	**0.0910**	0.0100	0.0300	0.0320	0.0360
o (10^-2)	--	--	--	--	--	--	0.0090	0.0083	0.0059	0.0058
p (10^-2)	--	--	--	--	--	--	**0.0021**	0.0340	**0.0280**	0.0050
q (10^-2)	--	--	--	--	--	--	0.0005	0.0006	--	--
r	--	--	--	--	--	--	--	--	**0.1212**	0.0058
s	--	--	--	--	--	--	--	--	**0.0262**	0.0001
t	--	--	--	--	--	--	--	--	--	0.0060

1. The first column is the same as that in Tab. 3.
2. *0 to 1* represents the software updating process from version 0 to version 1.

It can be seen that test cases generated by the proposed method are similar to the users' habit on the whole for covering software operations. However, some results don't meet expectations, which are thickened in Tab. 4. The main reason is that the random probability in fusing motifs configuration may not match the software usage in the next version. If a more accurate probability is added by an experienced engineer, this problem may be solved. Some special situations will also lead to an inaccurate result. For instance, the test cases from version 5 to 6 don't cover *d* since *d* is not operated in version 5. It is considered to use log file covering two or more previous versions instead of the last one.

To further verify usability, Markov model is structured by using X. Tian' method [13] for every time software is updated. Test cases with the same scale are generated through these models. The frequencies of the above operations are shown in Tab. 5.

Table 5. The frequencies of selected operations in test cases from Markov model

Op. (Unit)	0 to 1	1 to 2	2 to 3	3 to 4	4 to 5	5 to 6	6 to 7	7 to 8	8 to 9	9 to 10
a	0.0898	0.0901	0.0909	0.0907	0.0935	0.0996	0.0887	0.0754	0.0723	0.0747
b	0.0006	0.0007	.0011	0.0007	0.0007	0.0007	0.0009	0.0007	0.0005	0.0003
c	0.0318	0.0311	0.0352	0.0277	0.0302	0.0278	0.0275	0.0341	0.0345	0.0359
d (10^-2)	**0**	0.0008	0.0011	0.0011	**0.0018**	**0**	0.0015	0.0018	0.0015	**0.0016**
e	--	**0**	0.0054	0.0131	0.0192	0.0220	0.0211	0.0228	0.0275	0.0306
f	--	**0**	0.0003	0.0003	0.0004	0.0002	**0.0002**	**0.0006**	0.0002	0.0002
h	--	**0**	0.0048	0.0104	0.0207	0.0262	0.0260	0.0323	0.0336	0.0337
i	--	--	--	**0**	0.0007	0.0006	0.0006	0.0006	0.0008	0.0008
j	--	--	--	**0**	0.0008	0.0009	0.0009	0.0015	0.0014	0.0015
k (10^-2)	--	--	--	**0**	**0.0006**	0.0308	0.0288	0.0239	0.0196	0.0261
l (10^-2)	--	--	--	--	--	**0**	0.0012	0.0032	0.0183	0.0210
m (10^-2)	--	--	--	--	--	**0**	0.0361	0.0337	0.0240	0.0124
n (10^-2)	--	--	--	--	--	**0**	0.0045	0.0256	0.0334	0.0313
o (10^-2)	--	--	--	--	--	--	**0**	0.0078	0.0049	0.0057
p (10^-2)	--	--	--	--	--	--	**0**	0.0401	**0.0291**	0.0048
q (10^-2)	--	--	--	--	--	--	**0**	0.0006	**0.0019**	**0**
r	--	--	--	--	--	--	--	--	**0**	0.0032
0	--	--	--	--	--	--	--	--	**0**	0.0001
t	--	--	--	--	--	--	--	--	--	**0**

1. The first column is the same as that in Tab. 3.
2. The header is the same as that in Tab. 4.

Using Markov model from log analysis does not cover these new operations from software updating, leading to 0-frequency in test cases. But it can be seen that Markov model shows better stability. For example, there is no big deviation in (c, 1 to 2). Moreover, both the proposed method and Markov model can't avoid volatility, as some thickened data items shown. Generally, the proposed method obtains similar result as Markov model.

5. Conclusions and Perspectives

A reliability test cases generation method is proposed for software reliability test in software updating process. Log analysis is an effective automatic method for test cases generation. That said, the proposed method introduces temporal motifs idea and obtains the motifs in software usage log by motifs recovery algorithm. For solving the problem that log analysis can't cover new operations arising from software updating, motifs configuration is proposed and fused with motifs recovery to generate test cases. The proposed method is applied to an online smartphone application. Through contrasting the frequencies of some operations in log and test cases generated by the proposed method, the usability of the method is validated. As said in section 4.3, some limits exist in this method, such as volatility coming from huge changes in users' behaviors. Compared with Markov model, the usability is further verified. In the future works, we will do execution tests to give a further verification to the proposed method. More previous logs will be used to generate test cases for reducing the volatility.

Acknowledgements

This work has been supported by Project No. KKB517005534 of the National Key Technology Research and Development Program of China.

References

1. American National Standards Institute IEEE Standards Board, "IEEE Standard Dictionary of Measures to Produce Reliable Software" IEEE Std 982.1-1988
2. Mette Arleth, "Using Log-File Analysis for Testing Cartographic WebBased Applications" in *4 of the International Cartographic Conference*, Scientific and Technical Program Committee, 2008
3. M. Cavers and K. Vasudevan, "Spatio-temporal complex markov chain (scmc) model using directed graphs: earthquake sequencing," *Pure & Applied Geophysics*, vol. 172, no. 2, pp. 225-241, 2015
4. GB/T 11457-95. The national standard of the People's Republic of China - software engineering terms
5. P. Holme and J. Saramäki, Temporal Networks. *Physics Reports*, vol. 519, no. 3, pp. 97-125, 2016
6. H. Li, A. Zhao, D. Zhang and J. Zhang, "Research on building software usage model based on UML model," *International Journal of System Assurance Engineering & Management*, no. 10, pp. 1-9, 2017
7. Y. Lin, C. Wang, J. Wang and Z. Dou, "A Novel Dynamic Spectrum Access Framework Based on Reinforcement Learning for Cognitive Radio Sensor Networks," *Sensors*, vol. 16, no. 10, pp. 1-22, 2016 (DOI: 10.3390/s16101675)
8. Y. Lin, X. Zhu and Z. Zheng, "The individual identification method of wireless device based on dimensionality reduction and machine learning," *Journal of Supercomputing*, no. 5, pp. 1-18, 2017 (DOI: 10.1007/s11227-017-2216-2)
9. HD. Mills, M. Dyer and RC. Linger, "Cleanroom software engineering," *IEEE Software*, vol. 4, no. 4, pp. 19-25, 1987
10. R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii and U. Alon, "Network motifs: simple building blocks of complex networks," Science, vol. 298, no. 5594, pp. 824-827, 2002lo
11. J. Musa, "Operational profiles in software-reliability engineering," *IEEE Software*, vol. 10, no. 2, pp. 14-32, 1993
12. A. Paranjape, A. R. Benson and J. Leskovec, "Motifs in temporal networks," pp. 601-610, 2016
13. X. Tian, H. Li, and F. Liu, "Web Service Reliability Test Method Based on Log Analysis" in *IEEE International Conference on Software Quality, Reliability and Security Companion* (QRS-C), pp. 195-199, Prague, Czech Republic, July 2017
14. S. Zhang and J. Huang, "Reliability Test Cases Generation of Web Application Based On Log Analysis" in *International Conference on Computer Engineering, Information Science & Application Technology*, 2016