International Journal of Innovative Computing, Information and Control
Volume **11**, Number **5**, October **2015**

ICIC International ©2015 ISSN 1349-4198
pp. 1779–1793

EFFICIENT MINING OF HIGH UTILITY SOFTWARE BEHAVIOR PATTERNS FROM SOFTWARE EXECUTING TRACES

Haitao He[1,2], Tengteng Yin[1,2,*], Jun Dong[1,2], Peng Zhang[1,2] and Jiadong Ren[1,2]

[1]College of Information Science and Engineering
[2]The Key Laboratory for Computer Virtual Technology and System Integration of Hebei Province
Yanshan University
No. 438, West Hebei Ave., Qinhuangdao 066004, P. R. China
{ haitao; jdren }@ysu.edu.cn; *Corresponding author: yin_teng_teng@sina.com

Received April 2015; revised August 2015

Abstract. *In order to improve the understanding of the program, software behavior mining is very meaningful work. Finding desirable patterns can help the program maintainers comprehend the software adequately. These high utility software behavior patterns are some invocation patterns which shed light on program behaviors and capture unique characteristic of software traces. In this paper, a novel approach to mine high utility path patterns (HUPPMiner) from software executing traces is proposed. In HUPPMiner algorithm, a trace can contain repeated occurrences of interesting patterns due to loops. The loop times for a pattern are not much different in a software sequential pattern. Therefore, a continuous repetitive patterns eliminating algorithm is given firstly. Secondly, a novel structure called PL-Index-List storing both the location index and utility information of a pattern, and a HUPP-PL Tree storing the promising utility path patterns are put forward. Thirdly, an upper bound model and a new pruning strategy can be applied to prune the unpromising patterns early. Finally, the experimental results on synthetic datasets and real datasets show the proposed approach outperforms the traditional approaches in pruning effect and execution efficiency.*

Keywords: Software execution sequence, High utility path pattern, Software behavior, Repetitive patterns eliminating

1. **Introduction.** Improving software quality is an important goal of software engineering because software plays a critical role in businesses, governments and societies. Software behavior learning is one of the most important tasks in all stages of software development lifecycle [1]. These software behavior patterns can be used to detect the exception for a new software execution trace. For example, the functions of the mined software behavior patterns follow a specific order. If we find the functions in a new execution trace do not follow the order, it is possible that there exists an exception somewhere. As a result, how to mine desirable patterns from the large amounts of software traces is a very meaningful work.

From data mining viewpoint, each software executing trace can be considered as a sequence. Traditional frequent pattern mining algorithms consider only binary frequency values of items in transactions. Other information about items is not considered. To handle this, Yun and Leggett [2] proposed weighted sequential pattern mining. In addition, they designed an average weight function to evaluate the weight value of a pattern in a sequence. Nevertheless, a large number of candidate subsequences were still generated due to the upper-bounds of overestimating weighted values for the candidates. The IUA algorithm proposed in [3] is an efficient projection-based algorithm with an improved strategy

for weighted sequential pattern mining. It can prune more unpromising subsequences thanks to a tighter upper-bound in the mining process. As the weighted sequential pattern mining does not hold the property that the weight of the same item can be different in different sequences, the high utility pattern mining [4] model was proposed. However, since the downward-closure property in frequency-based mining cannot be directly used in utility mining, designing a proper upper bound that satisfies the downward-closure property is crucial to utility mining. Lan et al. [5] proposed a maximum utility measure. In addition, an efficient projection-based algorithm (PHUS) was also proposed by them. However, algorithm PHUS also suffers from producing and using projected databases. Yin et al. [6] presented a similar maximum utility concept to mine sequential patterns with high-utility. Moreover, they also proposed a tree-based mining approach named USpan. However, the USpan approach has to spend a great deal of execution time to traverse the LQS-Tree. Thus, how to reduce upper-bounds of utilities for subsequences in mining is quite important.

Software failure prediction is an important application of software behavior learning. Some sequential pattern mining algorithms also have been applied to software behavior mining. Feng and Chen [7] presented a multi-label software behavior learning algorithm named ML-KNN, which automatically classified a failure into one or more fault labels. Because a failing execution may be caused by several faults simultaneously, their approach can improve the effectiveness of software behavior learning significantly. Xia et al. [8] proposed a composite algorithm named MLL-GA which combined various multi-label learning algorithms by leveraging genetic algorithm. Their experiment results showed that MLL-GA had better efficiency than ML-KNN. Lo et al. [9] developed an algorithm called Closed Iterative Pattern MinER (CLIPER) that captured repetitive occurrences of the patterns within each trace and across multiple traces. Based on algorithm CLIPER, a method to classify software behaviors was put forward in [10]. They mined a set of discriminative features to detect failure. Li et al. [11] presented an approach of using the pattern position distribution as features to detect software failure. In their approach, the distribution of all patterns was used as features to train a classifier. Du et al. [12] put forward an approach that detected software failure by mining discriminative patterns from software behavior sequences. By selecting frequent closed unique iterative patterns as candidate pattern sets, they mined the discriminative binary and numerical patterns for sequence classification. A rule captures a constraint between its precondition and postcondition. The rules mined from software traces can reflect some interesting program behavior. Khoo [13] presented an algorithm to mine non-redundant significant recurrent rules from a set of program execution traces. Their algorithm also had good performance even at low support thresholds. To technically deepen research on iterative pattern mining, Lo et al. [14] introduced mining iterative generators. They referred to these rules as representative rules which could improve program understanding. To find common temporal rules, Lo et al. [15] developed an algorithm to mine rules of arbitrary lengths on traces which could be used to detect bugs. Acharya et al. [16] presented a framework to automatically extract frequent API patterns among user-specified APIs, directly from the source code from the perspective of static traces. These ordering rules existing between APIs govern the secure and robust operation of the system. Czibula et al. [17] proposed a method based on relational association rule mining for detecting faulty entities in existing software systems.

Considering that the items in software executing sequences are ordered and consecutive, the general sequential pattern mining algorithms are not applicable. The path sequential pattern has a property that the items appearing in the pattern must be adjacent with respect to the underlying ordering as defined in the pattern. Zhou et al. [18] proposed

a Two-Phase utility mining method to discover high utility path traversal patterns from weblog databases. Because their upper bound is loose, the number of candidates is large. Ahmed et al. [19] provided a very efficient algorithm for utility-based web path traversal mining by using a pattern growth sequential mining approach. However, longer patterns with less item utility may result in higher values. For this reason, Thilagu and Nadarajan [20] proposed an efficient algorithm to discover effective web traversal patterns based on average utility model. To reveal better results and resolve the problem occurring due to pattern length, the algorithm mined high average utility patterns rather than patterns with actual utility. Despite all this, the algorithm cannot deal with the sequences with both forward and backward references. Ahmed et al. [21] proposed a framework to mine high utility web access by two new tree structures. The algorithm avoids the level-wise candidate generation and test methodology.

Most of the existing high utility pattern mining and path traversal pattern mining algorithms do not consider that one item can appear multiple times in a sequence. As a consequence, these algorithms are not applicable for software executing sequences. In summary, our main contributions include the following.

- We extract software execution traces by tracing the executing process of software dynamically. In order to eliminate the influence of the loops, a continuous repetitive patterns eliminating algorithm is given basically.
- We propose two new data structures called PL-Index-List and HUPP-PL Tree respectively. By exploiting the two data structures, a new efficient algorithm for mining high utility path patterns based on pattern growth model is designed. These patterns shed light on some important software behaviors.
- We demonstrate through a comprehensive set of experiments to evaluate the efficiency and scalability of the proposed algorithm. In order to show the utility of the algorithm, we also analyzed the experimental results.

The remaining paper is organized as follows. Section 2 gives the definitions. The proposed mining algorithm HUPPMiner is stated in Section 3. Section 4 gives the experiments to analyze performances of the algorithm. The conclusion and future work are illustrated in Section 5.

2. **Definitions.** A software behavior can be viewed as a series of events. An event in turn corresponds to a unit behavior of interest. This can correspond to the execution of a statement, function, class, interface, etc. In this paper, we discuss from the function perspective. When software running, a series of events corresponding to a software behavior form an execution trace. Let I be a set of distinct functions. We denote a software executing trace T as $< E_{start}, E_2, \cdots, E_{end} >$. E_i (start $\leq i \leq$ end) is an element in T represented by a two tuple $(Sg, Fname)$ where Sg is the entrance-exit sign of function, namely a caller or callee in software. If it is a caller, Sg can be denoted as E, otherwise be X. $Fname$ is function name from a functions set I.

Definition 2.1. *STP (Software Trace Pattern). For a software trace, STP is defined as* $P(< E_i, \cdots, E_j >)$ *where* E_i *and* E_j *are the elements derived from sequence T, satisfying:* $E_i.Sg = E$, $E_i.Fname = E_j.Fname$, $E_j.Sg = X$.

In fact, the relationship between different STPs is very complex. Because of the existence of the loops in software execution trace, we mainly focus on the kind of continuous repetitive patterns.

Repetitive relationship: A software trace pattern $P_1(< s_1, s_2, \cdots, s_x >)$ is considered repetitive if another software trace pattern $P_2(< f_1, f_2, \cdots, f_y >) = P_1$ and P_2 is the

adjacent pattern of P_1. Then the repetitive relationship of them is denoted as P_1*. Here, P_1 is Continuous Repetitive Pattern of P_2, or vice versa.

Software execution trace contains entrance-exit sign which is neither complete nor intuitive. For this reason, the software executing sequences can be obtained by preprocessing the software execution traces. The set of input software executing sequences database under consideration is denoted by D.

Definition 2.2. *Path pattern is a subsequence of software executing sequence. It satisfies that items appearing in a software executing sequence containing the path pattern must be adjacent with respect to the underlying order as defined in the path pattern. A path pattern* $\alpha =< \alpha_1, \alpha_2, \cdots, \alpha_n >$ *is called a sub path pattern of another path pattern* $\beta = < \beta_1, \beta_2, \cdots, \beta_m >$ *if there exist integers* $1 \leq i_1 < i_2 < i_3 < i_4 \cdots < i_n \leq n \leq m$ *where* $\alpha_1 = \beta_{i_1}$, $\alpha_2 = \beta_{i_2}, \cdots, \alpha_n = \beta_{i_n}$. *We express this relation as* $\alpha \subseteq \beta$.

From Definition 2.2, we can know that path pattern does not allow gaps between different items.

Definition 2.3. *Join and Erasure. Given two patterns* $P_1(< a_1, \cdots, a_n >)$ *and* $P_2(< b_1, \cdots, b_m >)$, *if* $a_n = b_1$ *and the location index of* a_n *is equal to the location index of* b_1 *in a same software executing sequence, joining* P_1 *and* P_2 *will result in a longer pattern* $P_3(< a_1, \cdots, a_n, b_2, \cdots, b_m >)$. P_1 *is called prefix pattern and* P_2 *is suffix pattern. Considering a trace* $T(< E_1, E_2, \cdots, E_n >)$ *and an event* $E(E \in T)$, *the erasure of* T *to* E, *denoted by* $T - E$, *is defined as a new sequence* T_{sub} *formed from* T *where the events* $(E_1, E_2, \cdots, E)$ *are removed from* T.

Definition 2.4. *The utility of function* i *in a software execution sequence* S, *denoted as* $u(i, S)$, *is the product of* $iu(i, S)$ *and* $eu(i)$, *where* $u(i, S) = iu(i, S) \times eu(i)$.

Remark 2.1. *It should be noted that the* $eu(i)$ *in Definition 2.4 represents the external utility of function* i *which is the utility value of* i *in the utility table of* D. *It is the specific value assigned by a user to express the user's preference. This value reflects the importance of a function, which is independent of software executing sequences. If the user prefers function* e_j *to function* e_i, $eu(e_j)$ *is greater than* $eu(e_i)$. *The* $iu(i, S)$ *indicates the internal utility of function* i *which is the occurrence count of* i *in software execution sequence* S.

Definition 2.5. *The utility of sequence* S, *denoted as* $su(S)$, *is the sum of the utilities of all the items in* S *and the total utility of* D, *denoted as* $tsu = \sum_{S_i \in D} su(S_i)$, *is the sum of the utilities of all the sequences in* D. *The minimum utility threshold* ε *is given by the percentage of the total transaction utility values of the database. The minimum utility value can be defined as* $minutil = tsu \times \varepsilon$.

Definition 2.6. *For a path pattern* $X =< i_1, i_2, \cdots, i_n >$ *(*$X \subseteq S_i$ *and* $|X|$ *represents the length of* X*), its utility in a software execution sequence* S_i *denoted as* $u(X, S_i)$ *and the utility of* X *denoted as* $u(X)$ *are defined respectively as follows.*

$$u(X, S_i) = \left(\sum_{X \subseteq S_i} \sum_{i \in X} u(i, S_i) \right) / |X|. \tag{1}$$

$$u(X) = \sum\nolimits_{S_i \in D \wedge X \subseteq S_i} u(X, S_i). \tag{2}$$

The most challenging problem for high utility pattern mining is that the patterns utility does not hold the downward closure property. A pattern is a high utility pattern, but its super-pattern may be not. To maintain the downward closure property in high utility path pattern mining, we define a new upper bound for a path pattern.

Definition 2.7. *Suppose the maximum utility in software executing sequence S_i denoted as $msu(S_i)$ and the occurrence count in S_i of a path pattern X denoted as $oc(X, S_i)$. The utility upper-bound $puub_X$ of pattern X is represented below.*

$$puub_X = \sum\nolimits_{S_i \in D \wedge X \subseteq S_i} msu(S_i) \times oc(X, S_i). \tag{3}$$

A path pattern is high utility upper-bound path pattern if its upper bound value is greater than or equal to a predefined minimum utility threshold *minutil*.

Definition 2.8. *High Utility Path Pattern (HUPP). A path pattern X is high utility pattern if $u(X)$ is no less than a predefined minimum utility threshold minutil; otherwise, it is a low utility pattern.*

Lemma 2.1. *The utility upper-bound of a pattern maintains the downward closure property.*

Proof: Supposing y is a super-path pattern of path pattern x, then y cannot exist in any software executing sequences where x is absent. Therefore, the utility upper-bound $puub_x$ of x is the maximum upper-bound of utility value of y. Accordingly, if $puub_x$ is less than a predefined minimum utility threshold, then y cannot be a high utility path pattern. □

Lemma 2.2. *For a software executing sequence database D and a predefined minimum utility threshold called minutil, the set of high utility path patterns $HUPPS$ is a subset of high utility upper-bound path patterns $BHUPPS$.*

Proof: Let x be a high utility path pattern. According to Equation (2) and Equation (3), the actual utility $u(x)$ of x must be less than or equal to its utility upper-bound $puub_x$. Accordingly, if x is a high utility path pattern, then it must be a high utility upper-bound path pattern. As a result, x is a member of the set $BHUPPS$. □

3. **High Utility Path Pattern Mining from Software Executing Traces.** In this section we describe the process of our approach in detail. At first, we propose algorithm CRPE (continuous repetitive patterns eliminating), simplifying the software executing traces by eliminating the continuous repetitive patterns. Then an algorithm called HUPP-Miner (high utility path pattern mining) is put forward. The framework of our work can be seen as Figure 1. Software executing traces will be extracted from the process of software dynamic execution firstly. Secondly, due to the loop existing, we need to simplify these traces to eliminate continuous repetitive patterns. At last, high utility path patterns will be obtained by mining these simplified software executing sequences.

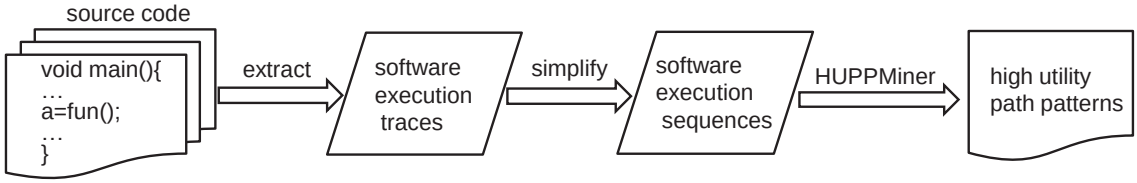

FIGURE 1. The framework of our work

3.1. **Continuous repetitive patterns eliminating.** Because of the loops existing in software executing trace, a trace can contain continuous repeated occurrences of some interesting patterns. Looping 10 times and looping 100 times for a pattern are not much different in a software sequential pattern. We only retain one occurrence for continuous repeated patterns. There is no need to eliminate the discontinuous repeated patterns, because the number of occurrences represents the importance of the patterns to some extent. Furthermore, if we eliminate continuous repetitive patterns, the experimental time can be reduced and the efficiency of the experiment can be improved greatly. The continuous repetitive patterns eliminating (CRPE) algorithm is described as Algorithm 1.

Algorithm 1: CRPE Algorithm

Input: software executing trace T
Output: software executing sequence S
1. **for** (int $i = 0$; $i < T$.length; $i = i + t$) **do**
2. **if** ($T(i)$.Sg==‘E’) **then**
3. **for** each element $y \in T - T(i)$ **do**
4. **if** (y.Sg==‘X’ and $T(i)$.Fname==y.Fname) **then**
5. replace $(Pi, P(T(i), y))$ in S; break; //the S is initialized to T at first call
6. **else** $t = 1$; break;
7. call procedure **Del-CRP(S, gap)** //Eliminate Continuous Repetitive Patterns, initialize gap to 1
8. call **CRPE(S)** repetitive until S is stable
9. **return** S

Procedure: Del-CRP
10. **if** (gap==1) **then** //gap represents the interval between repetitive patterns
11. delete other continuous repetitive patterns in S, retain the first one
12. **if** (gap!=1) **then**
13. delete other continuous repetitive patterns in S, retain the first one //we omit the elimination process
14. gap++
15. call procedure **Del-CRP(S,gap)** recursively until S is stable
16. **return** S

In lines 1 to 9, we patternize T to be S iteratively until S becomes stable. There are two processes mainly in the algorithm, the first is patternization (Line 5) and the second is elimination (Line 7). Lines 10 to 16 elaborate the process of eliminating continuous repetitive patterns.

Example 3.1. *Consider the following software sequence database D shown in Figure 2(a). In sequence S1, pattern* $< A, B, C >$ *continuous repeated twice and it needs to be eliminated one occurrence. Similarly, pattern* $< A, B, D >$ *in S2 and pattern* $< C >$ *in S3 also need to be eliminated. The result of elimination is shown Figure 2(b). As is shown in Figure 2(c), the external utility table includes six items, A, B, C, D, E and F, and their external utility values are 1, 4, 5, 3, 2 and 4 respectively. For S1 in Figure 2(b), the internal utility values of A, B, C and F are 2, 2, 2 and 1 orderly. The utility value of A can then be calculated as* $2 \times 1 (= 2)$. *All the other software executing sequences can be similarly processed. The utility values of the three sequences are shown as Figure*

Sid	software executing sequence
S1	< A, B, C, A, B, C, A, B, F ,C>
S2	<F, C, A, B, D, A, B, D, A, F>
S3	<C, C, C, E, A, B, D, E, A>

(a) The sample database

Sid	software executing sequence
S1	<A , B, C, A, B, F, C>
S2	<F, C, A, B, D, A, F>
S3	<C, E, A, B, D, E, A>

(b) The simplified database

item	A	B	C	D	E	F
Utility	1	4	5	3	2	4

(c) The external utility table

Sid	software executing sequence with utility	su	msu
S1	<A(2), B(8), C(10), A(2), B(8), F(4), C(10)>	44	10
S2	<F(8), C(5), A(2), B(4), D(3), A(2), F(8)>	32	8
S3	<C(5), E(4), A(2), B(4), D(3), E(4), A(2)>	24	5

(d) The sample sequences with utility

FIGURE 2. Database

2(d). In the third column of Figure 2(d) su shows the utility of each software executing sequence. The total utility of the database is 100. In the fourth column of Figure 2(d) msu shows the maximum utility of the sequence corresponding to the sequence serial number, for example, $msu(S1)$ is 10.

3.2. **High utility path pattern mining.** In section A, we propose a PL-Index-List structure to maintain the utility and location information of a pattern. In section B, we exploit the HUPP-PL Tree to describe the process of generating path patterns. A pruning strategy by using the PL-Index-List structure is also put forward. The algorithm HUPP Miner is described particularly in section C.

A. PL-Index-List Structure. To mine high utility path pattern, many previous algorithms directly perform on an original database. They have to compute the exact utilities of candidates by scanning the database. In the section, we propose a pattern index list structure based on location information (PL-Index-List) to maintain the utility and location information about a pattern. Firstly, the occurrence count of all items is calculated by the first database scan. If their upper-bound is less than $minutil$, these items are no longer considered in the subsequent mining process. Otherwise, construct the initial PL-Index-Lists for them by the second database scan. Each element in the PL-Index-List of pattern X contains five fields: sid, $eindex$, $nitem$, $util$ and msu.

- sequence serial number containing X
- the location index of the last item of pattern X
- the adjacent item of the last item of pattern X
- the utility of pattern X corresponding to sid
- the maximum utility of all the items in the sequence corresponding to sid

Suppose pattern Px is the combination of pattern P with item x, y is an item, and $Px.PL$ and $y.PL$ are the PL-Index-Lists of pattern Px and y. Algorithm 2 details how to construct the PL-Index-List for new path pattern Pxy combined by path pattern Px and y.

For each common software executing sequence S, the algorithm will generate an element E and append it to the PL-Index-List of Pxy. The sid field and the field msu of E are the sid and msu of S. The $eindex$ and $nitem$ of E are the $eindex$ and $nitem$ associated with S in the PL-Index-List of y respectively. The $util$ field of E is calculated according to the formula in line 5. Without scanning the database, the PL-Index-List of 2-length

Algorithm 2: Join Algorithm

Input: $Px.PL$, the PL-Index-List of Px; $y.PL$, the PL-Index-List of pattern y
Output: $Pxy.PL$, the PL-Index-List of Pxy
1. $Pxy.PL$ = NULL
2. **for** each element $Ex \in Px.PL$ **do**
3. **if** $\exists Ey \in y.PL$ **and** $Ex.sid == Ey.sid$ **then**
4. **if** $Ex.nitem == y$ **and** $(Ex.eindex + 1) == Ey.eindex$ **then**
5. $Exy =< Ex.sid, Ey.eindex, Ey.nitem, (Ex.util \times |Px| + Ey.util)/|Pxy|, Ex.msu >$
6. append Exy to $Pxy.PL$
7. **return** $Pxy.PL$

pattern Pxy can be constructed by the intersection of the PL-Index-List of Px and that of y. By comparing the *sid*, *eindex* and the *nitem* of the two PL-Index-Lists, we can judge whether the two patterns can be joined together.

Example 3.2. *The database D is shown as Figure 2(d), the items set* $I = \{A, B, C, D, E, F\}$. *Suppose the* ε *is 0.12, the minutil is 12 and then we no longer take item E into consideration after the first database scan. During the second database scan, the initial PL-Index-Lists are constructed. Figure 3 depicts the PL-Index-Lists of all the high utility upper-bound 1-length path patterns and partial 2-length path patterns. For example, consider the PL-Index-List of pattern* $< A >$. *In S1, it occurs two times and* $u(A, S1) = 2$, *so elements* $< 1, 1, B, 2, 10 >$ *and* $< 1, 4, B, 2, 10 >$ *are in the PL-Index-List of* $< A >$ *(*$< x1, x2, x3, x4, x5 >$ *means* $< sid, eindex, nitem, util, msu >$, *and 1 represents S1 for simplicity.). The rest can be figured out in the same manner. To construct the PL-Index-List of pattern* $< AB >$, *Algorithm 2 intersects the PL-Index-List of* $< A >$ *and that of* $< B >$, *which results in* $< 1, 2, C, 5, 10 >$, $< 1, 5, F, 5, 10 >$, $< 2, 4, D, 3, 8 >$ *and* $< 3, 4, D, 3, 5 >$.

<A>

1	1	B	2	10
1	4	B	2	10
2	3	B	2	8
2	6	F	2	8
3	3	B	2	5
3	7	ϕ	2	5

<B>

1	2	C	8	10
1	5	F	8	10
2	4	D	4	8
3	4	D	4	5

<C>

1	3	A	10	10
1	7	ϕ	10	10
2	2	A	5	8
3	1	E	5	5

<D>

2	5	A	3	8
3	5	E	3	5

<F>

1	6	C	4	10
2	1	C	8	8
2	7	ϕ	8	8

<AB>

1	2	C	5	10
1	5	F	5	10
2	4	D	3	8
3	4	D	3	5

<BD>

2	5	A	3.5	8
3	5	E	3.5	5

<CA>

1	4	B	6	10
2	3	B	3.5	8

<FC>

1	7	ϕ	7	10
2	2	A	6.5	8

FIGURE 3. The PL-Index-List of patterns

B. The HUPP-PL Tree and Pruning Strategy. The HUPP-PL Tree is used to store the path patterns and each pattern holds a PL-Index-List structure. The HUPP-PL Tree is an extension of the prefix tree. Given a set of items $I = \{f_1, f_2, \cdots, f_n\}$, the prefix tree can be constructed in the following way. Firstly, the root of the tree is created. Secondly, the $m(m \leq n)$ child nodes of the root representing m high utility upper-bound 1-length patterns are created respectively. Thirdly, for a node which represents pattern

$P(< f_s \cdots f_e > (1 \leq s \leq e < n))$, supposing the number of its different next items is k (only for these items which are included in high utility upper-bound 1-length patterns), the k child nodes of P representing patterns $< f_s \cdots f_e f_{e+1} >$, $< f_s \cdots f_e f_{e+2} >$, $\cdots$, $< f_s \cdots f_e f_{e+k} >$ are created. The third step is done repeatedly until all leaf nodes are created. An extension strategy is also proposed to ensure the efficiency of the algorithm.

Adjacent Path Pattern Extension Strategy. Considering that extension of software path patterns is consecutive, when a path pattern is extended, only node in next adjacency position of the path pattern needs to be considered. If node in next adjacency position of the path pattern is not a high utility upper-bound 1-length pattern, extension process will stop.

This strategy will avoid to produce some unpromising path patterns. To reduce the search space, we can exploit the *msu* value in the PL-Index-List of a path pattern. The sum of all the *msu* in the PL-Index-List is the upper-bound according to Definition 2.7. The path pattern can be extended further if the sum is no less than *minutil*; otherwise, we can prune the path pattern and its descendants safely according to Lemma 2.1.

Example 3.3. *Figure 4 depicts a HUPP-PL Tree representing all path patterns of Figure 2(d). Shaded rectangles represent candidates whose upper bound is no less than minutil. In Figure 4, patterns* $< AB >$ *and* $< AF >$ *are the 1-extensions of* $< A >$*, and* $< ABD >$ *is the 2-extension of* $< A >$*. For pattern* $< C >$*, its next items are A and E, but E has been removed when constructing the initial PL-Index-Lists for those upper bound 1-length patterns. According to our Adjacent Path Pattern Extension Strategy, we do not extend pattern* $< C >$ *by E. For pattern* $< B >$*, its three 1-extensions are* $< BC >$*,* $< BF >$ *and* $< BD >$*, and we only extend* $< BD >$ *because its upper bound is no less than minutil.*

C. The HUPPMiner Algorithm. The algorithm HUPPMiner is proposed to mine high utility path patterns from software executing sequence in this section. In Algorithm 3, firstly the high utility upper-bound 1-length path patterns are generated and are put in the set $BHUPP1$. Secondly, the initial PL-Index-Lists (PILs) for them are constructed.

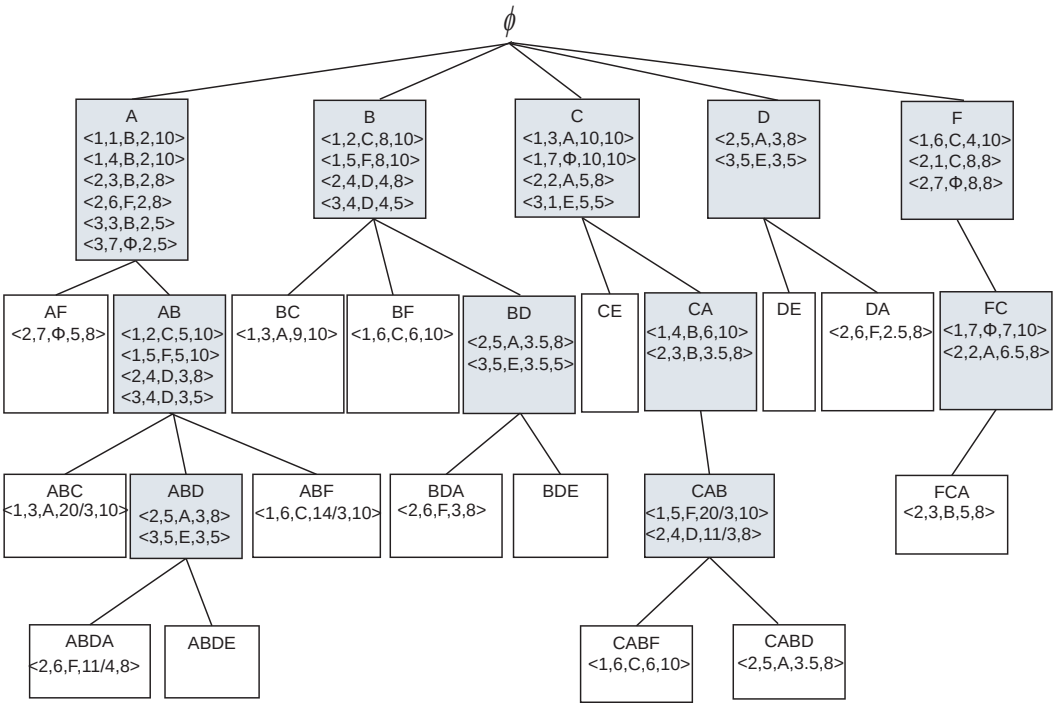

FIGURE 4. HUPP-PL Tree

Thirdly, a sub-procedure HUPP is devised to mine high utility path patterns efficiently from D. Lines 4 to 13 present the sub-procedure HUPP which is a recursive process. When the procedure HUPP is called in algorithm HUPPMiner firstly, the parameter $HUPPCS$ is the same as the $BHUPP1$. For each pattern x in $HUPPCS$, if the utility of x is no less than the minimum utility threshold $minutil$, then output it. If the upper bound value $puub_x$ of x is no less than $minutil$, the pattern x can be extended further. Lines 8 to 13 specify the extending process. The procedure HUPP intersects x and y if the item y is contained by the set of nitems of x.PL-Index-List. Algorithm Join(x, y) in line 11 is called to construct the PL-Index-List of pattern Pxy as stated in Algorithm 2. Finally, the set of PL-Index-Lists of all the 1-extensions of pattern x is recursively processed.

Algorithm 3: HUPPMiner Algorithm

Input: software executing sequence database D, a threshold *minutil*
Output: high utility path patterns
1. BHUPP1⟵finding the high utility upper-bound 1-length patterns
2. construct the initial PILs for all the patterns in BHUPP1
3. call procedure **HUPP(BHUPP1)**

Procedure: HUPP
Input: HUPPCS //the set of high utility path pattern candidates
4. **for** each pattern $x \in$ HUPPCS **do**
5. **if** $(u(x) \geq minutil)$ **then**
6. output x
7. **if** $(puub_x \geq minutil)$ **then** //it can be extended further
8. exUPPS = NULL
9. **for** each pattern $y \in$ BHUPP1 **do**
10. **if** (x.PL-Index-List.nitems.contains(y)) **then**
11. exUPPS = exUPPS + Join(x, y)
12. **if** (exUPPS != NULL) **then**
13. **HUPP(exUPPS)**

4. **Experiment.** A series of experiments were conducted to compare the performance of HUPPMiner with the existing utility path traversal pattern mining approaches Proposed [20] and EUWPTM [19] with different parameter values. As these two algorithms just consider the situation that the same item in a sequence is not repeated, to compare our algorithm with the two algorithms, we need process the datasets in advance. These algorithms were implemented in Java and the experiments were performed on 64 bit Windows 7 ultimate, Xeon CPU E5-2603 @1.80GHz, 8G Memory.

Experimental datasets. The synthetic datasets were generated by IBM data generator. To show the practical performance, two real datasets *Mushroom* and *Chess* were also used in the experiments and the dataset can be downloaded from FIMI Repository [22]. We also choose a real software *cflow* (for static analysis of C language code) to test the HUPPMiner. We get the experiment data of *cflow* with the help of *pvtrace*, *Gephi* and *Graphviz* on Linux. Similar to the previous algorithms, because the datasets do not provide the utility values of items, we randomly assign utility to each item. The utility distributions of items in datasets are generated from Gauss distribution ($\mu = 5$, $\sigma = 1.5$).

4.1. **Running time.** We varied the minimum utility threshold and tested the performance of the HUPPMiner algorithm on real datasets *Chess* and *Mushroom*. The minimum utility threshold range of 0.2-1.0% is used here. The dataset *Chess* has 3196 sequences, 75 items and the average length of sequence is 37. The dataset *Mushroom* has 8124 sequences, 119 items and the average length of sequence is 23. As shown in Figure 5 and Figure 6, we can find that the running time of the algorithms gets decreased by increasing the minimum utility threshold. The lower minimum utility threshold is, the larger performance difference between them becomes. For example, in Figure 5, when the minimum utility threshold varies between 0.2% and 1%, the three curves nearly cross at minimum utility threshold being 1%. When the minimum utility threshold is less than 1%, HUPPMiner performs the best, Proposed is second and EUWPTM is the last.

This is because EUWPTM and Proposed are Two-Phase utility mining algorithms. They produce candidate patterns firstly. Next, they have to scan the database again to find out the actual high utility path traversal patterns from the candidate patterns. As a result, if the candidate set is large, the efficiency of the two algorithms is not high. On the other hand, our algorithm exploits a sequential pattern growth mining approach and only needs two database scans. It can mine high utility path patterns without candidate generation, which avoids the costly generation and utility computation of candidates.

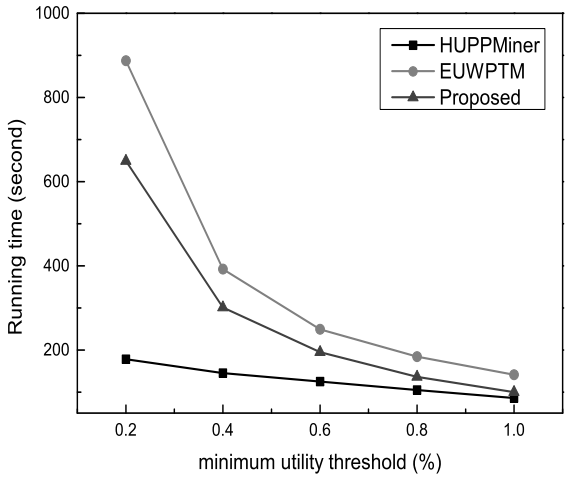

FIGURE 5. Running time on Chess

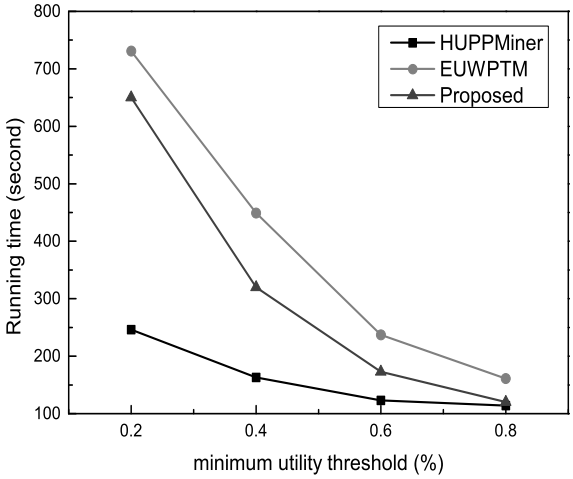

FIGURE 6. Running time on Mushroom

From the experimental results given in Figure 7 and Figure 8, it is also observed that the number of candidate patterns is decreasing gradually by increasing the minimum utility threshold. We understand that the number of candidate patterns is greatly reduced in the HUPPMiner algorithm compared to the two existing approaches. The results indicate the better performance of HUPPMiner. This benefits from superior pruning strategy of HUPPMiner, which is applicable for extension of software path patterns. HUPPMiner can prune more unpromising patterns thanks to a tighter upper-bound in the mining process. If the upper bound of a pattern is lower than minimum utility threshold, it is unnecessary to be extended. Considering maximum utility on the entire dataset, the other two algorithms produce a large number of candidates. In HUPPMiner, the upper bound of the patterns is the maximum utility in every sequence which is tighter than that in entire dataset.

4.2. **Scalability.** We studied the scalability of HUPPMiner algorithm on running time by varying the number of sequences in the dataset and average length of sequences. Figure

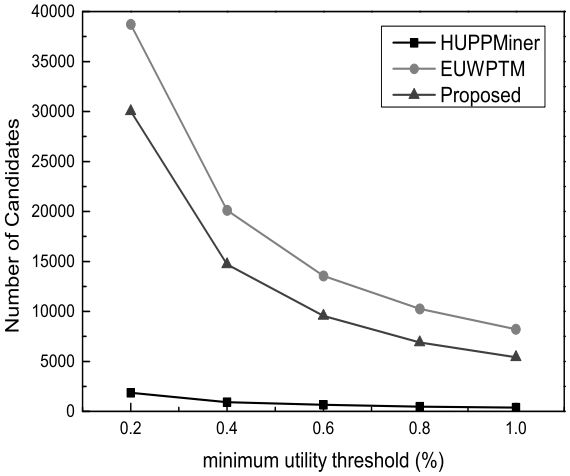

FIGURE 7. Number of candidates on Chess

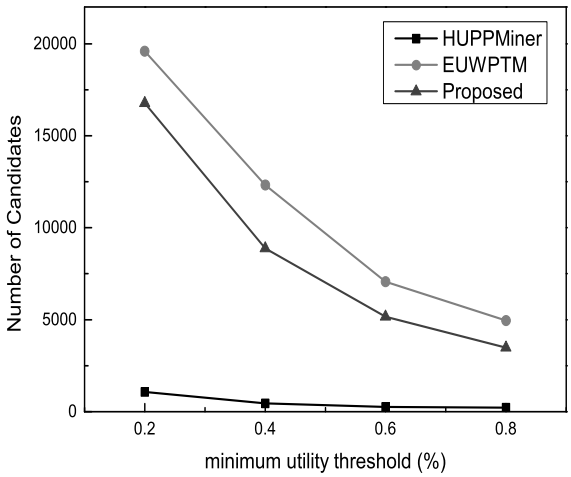

FIGURE 8. Number of candidates on Mushroom

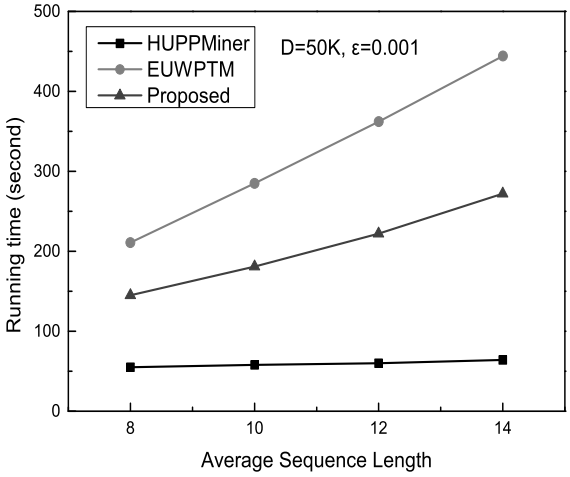

FIGURE 9. Running time with different sequence lengths according to $\varepsilon = 0.001$

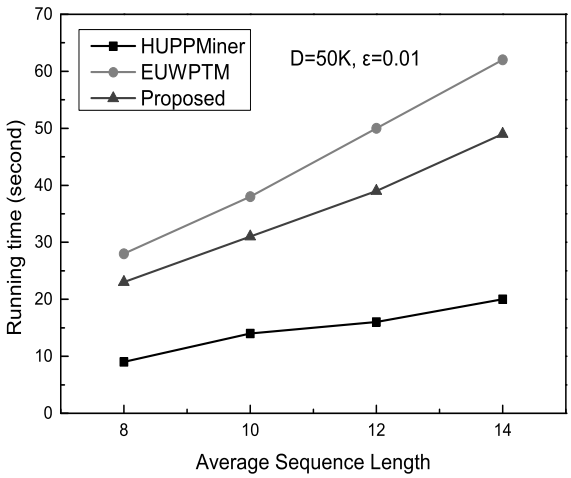

FIGURE 10. Running time with different sequence lengths according to $\varepsilon = 0.01$

9 and Figure 10 show the trend of the running time of three algorithms respect to different average lengths of sequences varying from 8 to 14 based on $|D| = 50000$ and minimum utility threshold 0.1% and 1% respectively. The running time of the three algorithms increase along with the average length of sequences increasing. In all case of different average lengths of sequences, HUPPMiner outperforms the other two algorithms. Especially when the average length of sequences is long, the time difference is more obvious. Figure 11 and Figure 12 show the results by varying the number of sequences from 10000 to 70000 with minimum utility threshold 0.1% and 0.2%. From the results, it is observed that as the database size increases, HUPPMiner outperforms the two existing algorithms.

The results indicate high accuracy and good scalability of HUPPMiner. This is because algorithms EUWPTM and Proposed need to calculate projection database iteratively. This operation is time-consuming for long sequences. The continuity characteristic of software path patterns is considered in HUPPMiner. Only when the upper bound of the next adjacency item is no less than the minimum utility threshold, we continue to extend. Moreover, HUPPMiner only builds PL-Index-Lists for potential high utility path

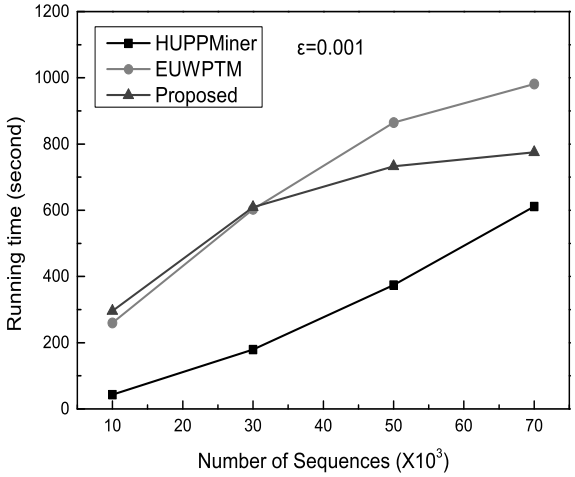

FIGURE 11. Running time with different database sizes according to $\varepsilon = 0.001$

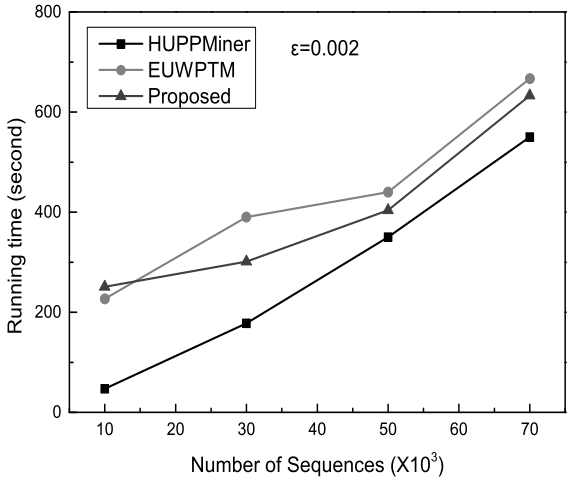

FIGURE 12. Running time with different database sizes according to $\varepsilon = 0.002$

TABLE 1. Top 10 high utility and frequent path patterns

high utility path patterns	frequent path patterns
$< 9, 10, 11, 22, 23 >$	$< 9, 10, 11, 22, 23 >$
$< 9, 10, 11, 12, 13 >$	$< 18, 8 >$
$< 9, 10, 11, 25, 26 >$	$< 6, 17, 27 >$
$< 1, 2, 3, 4, 7, 18 >$	$< 1, 2, 3, 4, 5 >$
$< 9, 10, 11, 12 >$	$< 9, 10, 11, 12 >$
$< 18, 6, 17 >$	$< 1, 2, 3, 4, 7, 18 >$
$< 1, 2, 3, 18, 6, 17 >$	$< 18, 6, 17 >$
$< 17, 27, 28 >$	$< 7, 8 >$
$< 1, 2, 3, 18, 6 >$	$< 17, 27, 28 >$
$< 1, 2, 3, 4, 5 >$	$< 1, 2, 3, 18, 6 >$

patterns. If the upper bound of the pattern is lower than minimum utility threshold, it is unnecessary to build PL-Index-List for the pattern.

4.3. **The discussion of results.** Table 1 shows the top 10 patterns discovered by the HUPPMiner and frequency-based algorithm on dataset $cflow$. The 1-length patterns are omitted. It is clear that path patterns found by the two algorithms are not always the same. The high utility path patterns $< 9, 10, 11, 12, 13 >$, $< 9, 10, 11, 25, 26 >$ and $< 1, 2, 3, 18, 6, 17 >$ are not frequent. On the other hand, path patterns $< 18, 8 >$, $< 6, 17, 27 >$ and $< 7, 8 >$ are dropped out by utility mining. The frequent path patterns $< 18, 8 >$, $< 25, 26 >$ and $< 7, 8 >$ are referred to $< parse\_variable\_declaration, expression >$, $< ident, lookup >$ and $< func\_body, expression >$ respectively. It indicates that the path pattern $< 9, 10, 11, 25, 26 >$ is high utility but not frequent. The reason is that functions *ident* and *lookup* have high utility but they are not frequent.

The mined patterns by the two methods are different because that only the occurrence number of the patterns is considered in frequency-based algorithm. The all items are treated equally. In that case, frequency-based algorithm will tend to mine short pattern. In the utility pattern mining, the utility can be defined according to some standards or features. The results are more desirable. Overall speaking, observed from our experiments, we realize that high utility path patterns are valuable, which can show the hidden behavior

patterns of software. These high utility path patterns in turn could be utilized to analyze the software characteristics.

5. **Conclusions and Future Work.** In this paper, we extend sequential pattern mining to consider repeated occurrences of pattern within sequences. In order to eliminate the loops in software executing traces, a continuous repetitive patterns eliminating algorithm is given at first. Next, we propose two new data structures called PL-Index-List and HUPP-PL Tree. The PL-Index-List provides not only utility and location information about patterns but also the upper bound value for pruning. The HUPP-PL Tree stores the promising utility path patterns. Then, a novel algorithm HUPPMiner is proposed to mine high utility path patterns from software executing sequences exploiting the two data structures. HUPPMiner can mine high utility patterns without candidate generation. We have demonstrated the performance of HUPPMiner in comparison with some other algorithms on various databases. Experimental results show that HUPPMiner has better efficiency, especially in terms of running time. In the future, we plan to apply our proposed algorithm in some real softwares.

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 P. R. China under Grant No. F2013203324, No. F2014203152 and No. F2015203326.

REFERENCES

[1] J. F. Bowring, J. M. Rehg and M. J. Harrold, Active learning for automatic classification of software behavior, *ISSTA*, pp.195-205, 2004.

[2] U. Yun and J. J. Leggett, WSpan: Weighted sequential pattern mining in large sequence databases, *Proc. of the 3rd International IEEE Conference on Intelligent Systems*, pp.512-517, 2006.

[3] G.-C. Lan, T.-P. Hong and H.-Y. Lee, An efficient approach for finding weighted sequential patterns from sequence databases, *Applied Intelligence*, vol.41, no.2, pp.439-452, 2014.

[4] C. F. Ahmed, S. K. Tanbeer and B. S. Jeong, A novel approach for mining high utility sequential patterns in sequence databases, *ETRI Journal*, vol.32, no.5, pp.676-686, 2010.

[5] G.-C. Lan, T.-P. Hong, V. S. Tseng et al., Applying the maximum utility measure in high utility sequential pattern mining, *Expert Systems with Applications*, vol.41, no.11, pp.5071-5081, 2014.

[6] J. Yin, Z. Zheng, L. Cao et al., USpan: An efficient algorithm for mining high utility sequential patterns, *KDD*, pp.660-668, 2012.

[7] Y. Feng and Z. Chen, Multi-label software behavior learning, *International Conference on Software Engineering*, vol.28543, no.1, pp.1305-1308, 2012.

[8] X. Xia, Y. Feng, D. Lo et al., Towards more accurate multi-label software behavior learning, *CSMR-WCRE, IEEE Computer Society*, pp.134-143, 2014.

[9] D. Lo, S.-C. Khoo and C. Liu, Efficient mining of iterative patterns for software specification discovery, *KDD*, pp.460-469, 2007.

[10] D. Lo et al., Classification of software behaviors for failure detection: A discriminative pattern mining approach, *KDD*, pp.557-566, 2009.

[11] C. Li, Z. Chen et al., Using pattern position distribution for software failure detection, *International Journal of Computational Intelligence Systems*, vol.6, no.2, pp.234-243, 2013.

[12] H. Du, C. Li and H. Wang, Mining multiple discriminative patterns in software behavior analysis, *UCAmI, Lecture Notes in Computer Science*, pp.511-518, 2014.

[13] S.-C. Khoo, Mining patterns and rules for software specification discovery, *PVLDB*, pp.1609-1616, 2008.

[14] D. Lo, J. Li, L. Wong et al., Mining iterative generators and representative rules for software specification discovery, *TKDE*, vol.23, no.2, pp.282-296, 2011.

[15] D. Lo, S.-C. Khoo and C. Liu, Mining temporal rules for software maintenance, *Journal of Software Maintenance and Evolution: Research and Practice*, pp.227-247, 2008.

[16] M. Acharya et al., Mining API patterns as partial orders from source code: From usage scenarios to specifications, *ESEC/FSE*, pp.25-34, 2007.

[17] G. Czibula, Z. Marian and I. G. Czibula, Detecting software design defects using relational association rule mining, *Knowledge and Information Systems*, vol.42, no.3, pp.545-577, 2015.
[18] L. Zhou, Y. Liu, J. Wang et al., Utility-based web path traversal pattern mining, *International Conference on Data Mining Workshops*, pp.373-380, 2007.
[19] C. F. Ahmed, S. K. Tanbeer, B. Jeong et al., Efficient mining of utility-based web path traversal patterns, *ICACT*, pp.2215-2218, 2009.
[20] M. Thilagu and R. Nadarajan, Efficiently mining of effective web traversal patterns with average utility, *ICCCS*, vol.1, no.4, pp.444-451, 2012.
[21] C. F. Ahmed, S. K. Tanbeer and B. Jeong, A framework for mining high utility web access sequences, *IETE Technical Review*, vol.28, no.1, pp.3-16, 2011.
[22] *Frequent Itemset Mining Dataset Repository*, http://fimi.ua.ac.be/, 2012.